HTLM5, CSS3, and JavaScript

Crash Course for Building Modern Responsive, Dynamic, and User-Friendly Websites for All Devices with The Aid of Practical Exercises

Sammie Smith

HTML5, CSS3, and JavaScript

Crash Course for Building Modern Responsive, Dynamic, and User-Friendly Websites for All Devices with The Aid of Practical Exercises

Sammie Smith

Copyright © 2023 **Sammie Smith**

All Rights Reserved

All rights reserved. No part of this book or any excerpts from it may be duplicated, reproduced, distributed, or transmitted in any form or by any means - mechanical, electronic, photocopying, recording, or otherwise - without the prior written consent of the publisher, except as permitted by American copyright law and fair use provisions.

Disclaimer and Terms of Use

The publishers and all individuals involved in the production of this

book have made every effort to ensure its quality, accuracy, and reliability. However, the authors and publishers cannot guarantee the correctness, applicability, suitability, or completeness of the contents. The information provided in this book is intended for educational and informational purposes only and should not be considered professional or legal advice. By reading and using the information in this book, you acknowledge and agree that you are solely responsible for your actions and decisions.

The authors, publishers, and contributors disclaim any liability for any loss or damages arising from the use of this book, including but not limited to direct, indirect, incidental, or consequential damages.

This book is intended for personal and professional guidance or consultation. Any examples or case studies provided are for illustrative purposes and should not be misinterpreted as a guarantee of results. All trademarks, and service marks in this book are the property of their respective owners, as they are a result of the owner's indebt research. References to specific products, services, or organizations do not imply endorsement or affiliation.

Printed in the United States of America.

If you found the information in this book useful, please kindly leave a review. Your feedback is greatly appreciated.

Thank you!

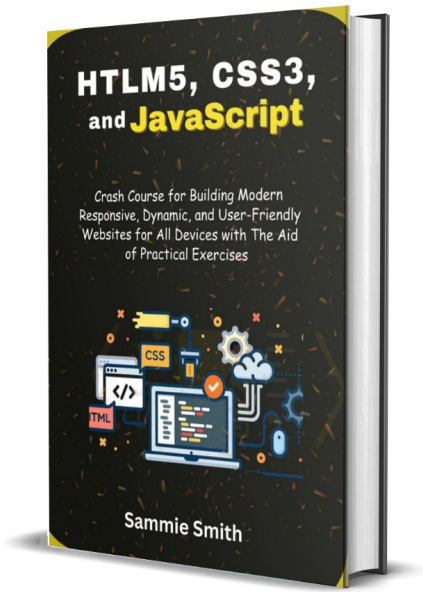

HTLM5, CSS3,
and JavaScript
Crash Course for Building Modern Responsive, Dynamic, and User-Friendly Websites for All Devices with The Aid of Practical Exercises
CSS
TML
Sammie Smith

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

CHAPTER ONE

GETTING STARTED WITH HTML

What is HTML?

What is New in HTML5
Why Learn HTML5
Choosing a Text Editor or IDE
Understanding HTML5 Semantic Markup
How is HTML used in Web Development?
Understanding the basic structure of an HTML document
Basic Tags and Attributes
Creating an HTML Document
Setting up an HTML Document
Adding Content to an HTML Document
Understanding HTML Tags, Attributes, and Values
Adding Comments to an HTML Document

CHAPTER TWO

HTML TEXT FORMATTING

Understanding HTML Headings
Understanding HTML Paragraphs
Understanding HTML Lists
Unordered Lists
Ordered Lists
Nested Lists
Understanding HTML links
Relative URLs
Linking to Specific Parts of a Page
How to Format Text using HTML Tags
How to Add Images to an HTML Document
How to Add Videos to an HTML Document
Using Audio in HTML5

CHAPTER THREE

UNDERSTANDING FORMS AND INPUT

Creating Forms
Text Inputs and Labels

Checkboxes and Radio Buttons

Select Menus and Option Groups

Text Areas and Buttons

Understanding Form Validation

CHAPTER FOUR

UNDERSTANDING TABLES AND LISTS

Creating Tables

Adding Rows and Columns

Styling Tables with CSS

CHAPTER FIVE

ADVANCED HTML5 FEATURES

Canvas and SVG graphics

How to Apply Canva and SVG Graphics

Understanding Web Storage

Understanding Geolocation

Understanding Web Workers

Understanding Drag and Drop

Understanding Web Sockets

Accessibility and SEO

Making your HTML5 Code Accessible

Using Semantic Markup

Applying Semantic Markup

Optimizing for Search Engines

CHAPTER SIX

HTML PRACTICAL EXERCISES

Exercise 1: Creating a Login Page

Exercise 2: Creating a Registration Page

Exercise 3: Create a Simple Personal Portfolio Website

CHAPTER SEVEN

INTRODUCTION TO CSS

What is CSS?

How is CSS used in Web Development?

How CSS works with HTML

Understand the Difference Between CSS and HTML

Benefits of using CSS

Basic CSS Syntax

Writing CSS rules

Selectors and Declarations

Comments in CSS

External and Internal CSS

Linking an External CSS

CHAPTER EIGHT

CSS STYLE PROPERTIES

Text Properties

What are CSS Text Properties?

Applying Text Properties

Font Properties

Applying Font Properties

Color Properties

Applying Color properties

Background Properties

CSS Box Model

Applying CSS Box Model

Understanding the Box Model

Margin, Border, and Padding

Width and Height

CSS Layouts

CSS Floats

Applying Float

Positioning

Applying Positioning

Display
Applying Display
Understanding CSS Grid
Applying CSS Grid
CSS Animations
Applying CSS Animations
CSS transitions
Animating with Keyframes
Undesratanding Flexbox
Applying Flexbox
CSS Variables
Scalable Vector Graphics (SVG)
Writing Efficient CSS
Debugging CSS
The Future of CSS

CHAPTER NINE

HTML AND CSS PRACTICAL EXERCISES

Exercise 1: Design a simple login page with HTML and CSS
Exercise 2: Build a simple landing page for a product with HTML and CSS.
Exercise 3: Create a responsive pricing table with HTML and CSS

CHAPTER TEN

RESPONSIVE WEB DESIGN

Introduction to Responsive Design
Best Practices for Designing Responsive Websites
Tools and Technologies used to Create Responsive Website
Understanding Media Queries
Applying Media Queries
Understanding Mobile-first Design Approach
Applying Mobile-first Design
Responsive Frameworks

Common Challenges and Solutions Associated with Responsive Web Design

Optimizing Images and Managing Complex Layouts

CHAPTER ELEVEN

RESPONSIVE WEBSITES PRACTICAL EXERCISES

Exercise 1: Design a responsive restaurant website

Exercise 2: Create a Responsive Photographer Portfolio

Exercise 3: Develop a Responsive E-commerce Website

CHAPTER TWELVE

GETTING STARTED WITH JAVASCRIPT

WHAT EXACTLY IS JAVASCRIPT

HISTORY OF JAVASCRIPT

FEATURES OF JAVASCRIPT

DIFFERENCES BETWEEN JAVASCRIPT AND OTHER PROGRAMMING LANGUAGES

UNDERSTANDING OBJECT-ORIENTED PROGRAMMING

SETTING UP JAVASCRIPT DEVELOPMENT ENVIRONMENT

HOW TO CONFIGURE VISUAL STUDIO CODE TO RUN JAVASCRIPT

CHAPTER THIRTEEN

JAVASCRIPT BASICS

JAVASCRIPT VARIABLES

Strings:
Numbers:
Booleans:
Objects:
Arrays:

DECLARING VARIABLES

WORKING WITH STRINGS, NUMBERS, AND BOOLEAN VALUES

WORKING WITH STRINGS

Declaring a String Variable

Accessing Individual Characters in a String
Concatenating Strings

WORKING WITH NUMBERS
Declaring a Number Variable

WORKING WITH BOOLEANS
Using Boolean Operators

UNDERSTANDING OBJECTS AND ARRAYS
Understanding Objects
Declaring Objects
Understanding Arrays
Declaring Arrays

CHAPTER FOURTEEN

OPERATORS AND EXPRESSIONS

ARITHMETIC OPERATORS
Addition (+):
Subtraction (-):
Multiplication (*):
Division (/):
Modulo (%):

COMPARISON OPERATORS
Equal to (==):
Not Equal to (!=):
Greater than (>), than (<):
Greater than or equal to (>=), less than or equal to (<=):

LOGICAL OPERATORS
AND Operator (&&):
OR Operator (||):
NOT Operator (!):

CONDITIONAL STATEMENTS (IF/ELSE)
The if Statement:
The Else Statement:
The else if statement:

THE SWITCH STATEMENT

CHAPTER FIFTEEN

FUNCTIONS AND SCOPE

DECLARING AND CALLING FUNCTIONS

PARAMETERS AND ARGUMENTS

RETURN VALUES

SCOPE AND CLOSURES

Global Scope

Closures

CHAPTER SIXTEEN

ARRAYS AND LOOPS

WORKING WITH ARRAYS AND ARRAY METHODS

The For Loop

The While Loop

The Do-While Loop

The ForEach Method

CHAPTER SEVENTEEN

OBJECTS AND CLASSES

CREATING AND WORKING WITH OBJECTS

Object Methods

CREATING CLASSES

CONSTRUCTORS AND PROTOTYPES

CHAPTER EIGHTEEN

DOM MANIPULATION

INTRODUCTION TO THE DOCUMENT OBJECT MODEL (DOM)

SELECTING AND MANIPULATING ELEMENTS

getElementById

querySelector

querySelectorAll

MANIPULATING ELEMENTS

innerHTML

textContent

setAttribute

EVENT HANDLING

PRACTICAL IMPLEMENTATION OF DOM MANIPULATION

CHAPTER NINETEEN

ASYNCHRONOUS JAVASCRIPT

UNDERSTANDING ASYNCHRONOUS PROGRAMMING

CALLBACK FUNCTIONS

UNDERSTANDING PROMISES

ASYNC/AWAIT

CHAPTER TWENTY

AJAX AND APIS

INTRODUCTION TO AJAX

WORKING WITH APIS

Fetching data from an API using the Fetch API
Sending data to an API using the Fetch API
Using a third-party API wrapper library like Axios
Implementing APIs in a Practical Exercise

FETCHING DATA WITH AJAX

DISPLAYING DATA ON A WEBPAGE

CHAPTER TWENTY ONE

UNDERSTANDING ES6

INTRODUCTION TO ES6

NEW FEATURES AND SYNTAX

WORKING WITH MODULES

FUTURE UPDATES TO JAVASCRIPT

COMMON MISTAKES TO AVOID WHEN WRITING JAVASCRIPT CODE

Common Mistakes to Avoid

BEST PRACTICES FOR WRITING CLEAN, MAINTAINABLE CODE

CHAPTER TWENTY TWO

PRACTICAL EXERCISES

EXERCISE 1: CREATING A SIMPLE APP TO MANAGE TASKS AND

TO-DO LISTS

EXERCISE 2: CREATING A CALCULATOR APP TO PERFORM SIMPLE AND COMPLEX MATHEMATICAL OPERATIONS

EXERCISE 3: CREATING A SIMPLE GAME USING JAVASCRIPT

DEPLOYMENT AND HOSTING OPTIONS.

UNDERSTANDING SAVERAL CHEAPER PLATFORMS TO HOST YOUR WEBSITES

HOW TO HOST YOUR WEBSITE

CONCLUSION

INDEX

INTRODUCTION

In today's digital landscape, where smartphones, tablets, and a multitude of devices have become an integral part of our lives, the need for responsive web design has never been more crucial. A responsive website seamlessly adapts to various screen sizes, providing an optimal user experience and ensuring that your content is accessible to all.

This book is a comprehensive guide that takes you on a journey to master the art of building responsive websites using the three pillars of modern web development: HTML, CSS, and JavaScript. Whether you are a seasoned developer or just starting your web development journey, this book will equip you with the essential knowledge and skills to create stunning and user-friendly responsive websites.

We will begin by exploring HTML (HyperText Markup Language), the backbone of the web. You will learn how to structure your web pages using semantic HTML, ensuring that your content is well-organized and accessible to both humans and search engines. With a solid foundation in HTML, we will then delve into the realm of CSS (Cascading Style Sheets) and discover the power of styling. You will uncover the secrets of responsive design, leveraging CSS media queries, flexible layouts, and fluid grids to create visually appealing and adaptable web interfaces.

But our journey doesn't end there. To truly unlock the potential of responsive web development, we will harness the power of JavaScript. You will learn how to use JavaScript to create dynamic and interactive elements on your website. From implementing responsive navigation menus to building interactive forms and integrating third-party libraries, you will gain the skills to make your websites come alive and engage users across all devices.

Throughout this book, we will not only cover the technical aspects of HTML, CSS, and JavaScript but also delve into best practices, performance optimization techniques, and cross-browser compatibility. You will learn how to optimize your website's performance, handle different screen sizes, and ensure a seamless user experience regardless of the device being used.

By the end of this book, you will have a deep understanding of HTML, CSS, and JavaScript, and the ability to create stunning, user-friendly, and responsive websites that adapt flawlessly to any device. Whether you are building a personal blog, an e-commerce platform, or a corporate website, this knowledge will empower you to deliver an exceptional user experience and stay ahead in today's competitive digital landscape.

So, if you are ready to embark on a transformative journey into the world of responsive web development, "Building Responsive Websites with HTML, CSS, and JavaScript: Unleashing the Power of the Web Trio" is your comprehensive companion. Let's dive in and unlock the full potential of the web trio to create remarkable and responsive digital experiences that leave a lasting impact on your audience.

CHAPTER ONE

GETTING STARTED WITH HTML

In this chapter, we will introduce the basics of HTML, including tags, attributes, and elements. We will explain how to create an HTML document, set up a basic structure, and add content.

What is HTML?

HTML stands for Hypertext Markup Language. It is a standard markup language used to create web pages and other types of digital content that can be displayed on the Internet. HTML provides a way to structure and format content such as text, images, and multimedia elements, allowing web browsers to render them in a visually appealing and organized manner.

HTML works by using a set of tags and attributes to define the structure and appearance of web page content. These tags are enclosed in angle brackets (< >) and specify the type of content they contain. For example, the <h1> tag is used to define a heading, while the <p> tag is used to define a paragraph. Attributes are used to provide additional information about a tag,

such as its class, id, or style.

Web developers use HTML in conjunction with CSS (Cascading Style Sheets) and JavaScript to create dynamic, interactive web pages. HTML5 is the latest version of the HTML specification and includes new features such as video and audio elements, as well as improved support for mobile devices.

What is New in HTML5

HTML5 introduced several new features and improvements over previous versions of HTML. Some of the key new features include:

- **Video and Audio Features:** Audio and video elements are two essential new features of HTML5. It makes it possible for site designers to add a video or audio clip to their pages. HTML5's video element can be styled with CSS and CSS. Changes can be made to borders, opacity, gradients, reflections, transitions, transformations, and even animations. By making the code available for use in embedding their films on other websites, YouTube also announces video embedding. It helps the web become more and more interactive with multimedia. To embed any audio clip into a webpage on the Internet, HTML5 provides a new element called the audio tag that is available for use.
- **Header and Footer:** The need for a div> tag to divide the two portions is gone thanks to these new tags. The header is at the top of the page, while the footer is at the bottom. If you utilize the HTML5 elements header and footer, the browser will understand what to load first and what to load later.
- **New input tag types included**: An outdated attribute called "input" has been given new life in HTML by taking on new values like "email," "month," "number," "range," "search," "tel," "color," "week," "URL," "time," "date," "DateTime-local," and so on. The input tag may now contain the following new values:

    1. **ContentEditable:** It's a feature that enables content modification by the user. It makes it simple to tell if

what you see is what you get. You will be able to edit the content by clicking on it.

2. **Progress**: This tag is used to keep track of a job's progress as it is finished. The JavaScript progress tag is compatible with it. It has a progress bar-like appearance.
3. **Section**: A document can be divided into sections or components using the section tag. For instance, an article might have several sections, including a header, footer, section for the main content, and the newest news, among others.
4. **Main:** The main tag is used to summarize the page's main content. A document may include just one main tag, and this tag may not be contained within the article aside footer, or header tags. There is no header, footer, or navigation bar.

- **Figure and figcaption:** The addition of a figure and a caption to a document was not before possible. With the advent of the figure and figcaption tags, it is now semantically possible to embed an image together with its explanation into a website. The syntax is as follows:

```
<figure>
    <img src="image/image-1.jpg" alt="About ADMEC" />
        <figcaption>
            <p>This is our institute </p>
        </figcaption>
</figure>
```

- **Placeholders:** Before, we had to utilize a little JavaScript to create placeholders for text fields. Sure, you can first alter the value property as you see fit, but if the user deletes that information and then clicks away, the input will be left empty. Using the placeholder property corrects this.
- **Preload Videos**: It's an excellent feature for sharing videos. It explains how to upload the video and how long it will take the website to load. This notifies the browser about the improvements made to the user experience of the webpage. Although it's not a

requirement, you should nonetheless add this feature. It contributes to a more realistic representation of the page.

- **Controlling the display:** The behavior of elements is determined by the display property. If this attribute is not specified, the default values are applied.
- **Regular Expressions**: To add a certain pattern as an input, we may use a regular expression. For instance, the most common pattern is [A-Za-z] 5,11. Both capital and lowercase letters are acceptable. Additionally, it says that the minimum and maximum character lengths are five and eleven, respectively.
- **Adaptability:** Since its creation, HTML5 has significantly contributed to a website's ability to offer the best accessibility features. As a result, the website's usage has been simplified. The accessibility features of HTML5 allow users with virtually any type of disability, including vision impairment, color blindness, impaired vision, blindness, and more, to access websites. For instance, the best illustration of providing accessibility in forms is validation. Labels must be seen clearly.
- **Inline components**: These inline items are very helpful in keeping code current:

    1. **mark:** It highlights items that have been in some way marked.
    2. **Time**: The website's current time and date are shown using this.
    3. **Meter**: It shows how much space is left available on the storage disk.
    4. **Progress bar**: The progress bar lets you monitor the status of a task you have been given.

- **Support for Dynamic Pages:** Nowadays, dynamic and interactive websites are preferred over static ones. The website has a dynamic vibe because of a few factors:

    1. **Mark**: It highlights items that have been in some way marked.
    2. **Time**: The website's current time and date are

shown using this.

3. **Meter**: It shows how much space is left available on the storage disk.
4. **The progress:** The progress bar lets you monitor the status of a task you have been given. Other factors also contribute to the website's dynamic nature.

- **Email as a property:** The browser automatically takes the instruction from the code to produce an email in the correct and valid format when we specify the type of email in a form. Earlier browsers did not make this possible.
- **Cryptographic Nonces**: In this most recent version of HTML, we may now apply cryptographic nonces to all styles and scripts. Our standard practice is to use the nonce attribute inside the script and style elements. In essence, this nonce tag generates a random integer that is only used once. As a result, the page is updated each time it is refreshed. It's a great feature since it can be used to increase the content of the page's security, allowing the website to declare and choose a certain script or style.
- **Reverse Links:** The rev attribute for reverse links is back in use with HTML 5.1. In essence, it gives web users the ability to use the link and anchor tag elements once more. It also explains the relationship between the linked document and the current document in reverse.
- **Photographs with a width of Zero**: Now, website developers can adjust the width of the images to zero. When there is no need to show them to customers, such as when monitoring picture files because they would otherwise take up more space, this capability is helpful. Utilizing images with a zero width and a blank alt tag is advised.
- **Canvas in HTML5:** A canvas is a rectangular area that can be used for pixel-level actions such, among other things, drawing a line, box, or circle, or running images. The support for canvas regions in HTML5 has been added. Below is a sample of a program's code.

```
<canvas id="myCanvas" width="200" height="100">
    Fallback content, when canvas is not supported by the browser.
    </canvas>
```

Why Learn HTML5

There are many reasons why learning HTML5 is a valuable skill for anyone interested in web development or design. Here are a few of the main reasons:

- **Essential for building websites**: HTML5 is the foundation of all modern websites. Understanding HTML5 is essential for anyone who wants to create, edit, or customize websites.
- **Improved web standards**: HTML5 is designed to improve web standards and make the web more accessible and user-friendly. By learning HTML5, you can help ensure that your websites are built to these standards.
- **Better multimedia support**: HTML5 introduces new elements and features for embedding multimedia content, such as videos, audio, and images. This makes it easier to create rich and engaging websites.
- C**ross-platform compatibility**: HTML5 is supported across all major web browsers and devices, including desktops, laptops, tablets, and smartphones. This makes it easier to create websites that work seamlessly across different platforms and devices.
- **Web application development**: HTML5 also provides features and APIs for building advanced web applications, such as local storage, geolocation, web sockets, and more. These features can help you create more interactive and engaging web applications.
- **High demand for HTML5 developers**: There is a high demand for web developers who are proficient in HTML5, especially those who also have experience with other web development technologies like CSS and JavaScript. Learning HTML5 can open up many job opportunities in the tech industry.

Choosing a Text Editor or IDE

Choosing a text editor or Integrated Development Environment (IDE) is an important step when setting up your HTML5 development environment. Here are some factors to consider when choosing a text editor or IDE:

- **Features**: Look for a text editor or IDE that has features that will make your development process easier, such as syntax highlighting, auto-completion, code folding, and code snippets.
- **Ease of use**: Choose a text editor or IDE that is easy to use and has an intuitive user interface. The text editor or IDE should be easy to navigate and allow you to quickly find the tools you need.
- **Compatibility**: Make sure the text editor or IDE you choose is compatible with your operating system and other tools you plan to use, such as your web browser and web server.
- **Community support**: Choose a text editor or IDE that has a large and active community of users, as this can be a valuable resource for getting help, finding plugins and extensions, and sharing your work.
- **Price**: Consider the cost of the text editor or IDE. Some text editors, like Sublime Text and Atom, offer free versions with limited features, while others, like Visual Studio Code and WebStorm, require a paid license for full access to all features.

Here are some popular options for text editors and IDEs for HTML5 development:

- **Visual Studio Code**: A free and open-source IDE developed by Microsoft, with support for HTML, CSS, and JavaScript.
- **Sublime Text:** A lightweight text editor with a free version and a paid license for additional features.
- **Atom**: A free and open-source text editor developed by GitHub, with support for HTML, CSS, and JavaScript.
- **Brackets**: A free and open-source text editor developed by Adobe, with support for HTML, CSS, and JavaScript.

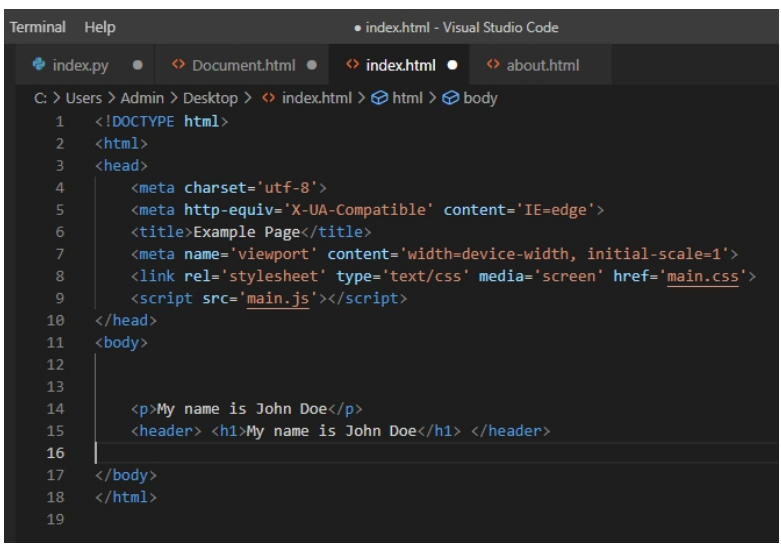
- **WebStorm**: A paid IDE developed by JetBrains, with support for HTML, CSS, and JavaScript.

Ultimately, the choice of a text editor or IDE is a personal preference, and it's important to find the one that works best for you and your workflow. Every exercise contained in this book is deployed using Visual Studio Code.

Understanding HTML5 Semantic Markup

Semantic markup is the practice of using HTML tags that accurately describe the meaning and purpose of the content they contain. In other words, semantic markup is the use of HTML tags to give meaning to the content of a web page, rather than just for visual presentation.

For example, consider the following two HTML code snippets:



In the first example, the content is simply marked up as a paragraph. In the second example, however, the content is marked up using a header and a heading element, which gives it more meaning and context.

By using semantic markup, web developers can make their web pages more accessible to all users, including those who use assistive technologies, such as screen readers. Semantic markup also helps search engines better understand the content of a web page, which

can improve search engine rankings.

Some examples of semantic HTML tags include:

- <**header**>: Used to define the header of a section or web page
- <**nav**>: Used to define a section of navigation links
- <**article**>: Used to define a standalone piece of content, such as a blog post or news article
- <**section**>: Used to define a section of related content
- <**footer**>: Used to define the footer of a section or web page
- <**aside**>: Used to define content that is related to the surrounding content, but not a part of it

Using semantic markup can help make your HTML code more meaningful, readable, and accessible.

How is HTML used in Web Development?

HTML is a fundamental part of web development and is used to create the structure and content of web pages. When a user visits a website, their web browser retrieves the HTML code from the server and interprets it to display the content on the screen.

HTML is used in web development in the following ways:

- **Creating the structure of web pages**: HTML provides the basic structure of a web page by defining headings, paragraphs, lists, tables, and other structural elements.
- **Adding content to web pages**: HTML is used to add text, images, videos, audio files, and other types of content to web pages.
- **Creating links**: HTML allows developers to create links between web pages, enabling users to navigate between them.
- **Building forms**: HTML is used to create web forms that allow users to submit data to a website, such as contact information or user login details.
- **Adding metadata**: HTML allows developers to add metadata to web pages, such as title tags, meta descriptions, and keywords. This information is used by

search engines to index and rank web pages.

Note that HTML is a critical component of web development, and knowledge of HTML is essential for creating functional and attractive web pages.

Understanding the basic structure of an HTML document

here's a brief overview of the basic structure of an HTML document:

- <**!DOCTYPE html**>: This is the document type declaration, which tells the web browser which version of HTML the document is written in. For HTML5, this should always be the first line of your document.
- <**html**>: The <**html**> tag is used to define the beginning and end of the HTML document. All the content of the document should be enclosed within these tags.
- <**head**>: The <**head**> tag contains meta-information about the document such as the title of the page, links to CSS stylesheets, and other metadata that is not displayed in the browser.
- <**title**>: The <**title**> tag is used to define the title of the document. This title appears in the browser's title bar and is also used by search engines to describe the page.
- <**body**>: The <**body**> tag is used to define the main content of the document. This is where you put all of the visible content that will be displayed in the browser.
- <**h1**>, <**h2**>, <**h3**>, <**h4**>, <**h5**>, <**h6**>: These are the heading tags, which are used to define headings and subheadings within the document. <**h1**> is the largest heading and <**h6**> is the smallest.
- <**p**>: The <**p**> tag is used to define paragraphs of text.
- <**a**>: The <**a**> tag is used to create hyperlinks to other web pages or other sections within the same document.
- <**img**>: The <**img**> tag is used to insert images into the document.

- <**ul**> and <**li**>: These are the unordered list and list item tags, respectively. They are used to create bulleted lists.
- <**ol**> and <**li**>: These are the ordered list and list item tags, respectively. They are used to create numbered lists.
- <**div**>: The <**div**> tag is used to group related elements in the document. It is often used in conjunction with CSS to apply styles to a group of elements.

That's a quick overview of the basic structure of an HTML document. Keep in mind that there are many more tags and attributes available in HTML, but these are some of the most commonly used ones.

Basic Tags and Attributes

HTML5 includes a wide variety of tags and attributes that you can use to create content and structure web pages. Here are some of the most common HTML tags and attributes:

Tags:

- <**html**>: The root tag that wraps around all other content on the web page.
- <**head**>: Contains meta-information about the web page, such as the page title and links to external stylesheets and scripts.
- <**body**>: Contains the main content of the web page.
- <**h1**> to <**h6**>: Used to create headings of varying levels of importance.
- <**p**>: Used to create paragraphs of text.
- <**a**>: Used to create hyperlinks to other web pages or resources.
- <**img**>: Used to insert images into the web page.
- <**ul**> and <**ol**>: Used to create unordered and ordered lists, respectively.
- <**li**>: Used to create list items.
- <**div**> and <**span**>: Used to group content and apply styles to it.
- <**form**>: Used to create a form for user input.

Attributes:

- **id**: Used to uniquely identify an element on the web page.
- **class**: Used to group elements together and apply styles to them.
- **src**: Used to specify the URL of an image or other resource.
- **alt**: Used to provide alternative text for images, which is used by screen readers and search engines.
- **href**: Used to specify the URL that a hyperlink points to.
- **target**: Used to specify where the linked resource should be displayed (e.g. in a new window or tab).
- **type**: Used to specify the type of input in a form field (e.g. text, password, checkbox, radio button).

These are just a few of the many HTML tags and attributes available. As you become more familiar with HTML, you'll discover additional tags and attributes that can be used to create more complex and interactive web pages.

Creating an HTML Document

Here are the basic steps to create an HTML document:

- Open a text editor on your computer like Visual Studio.
- Start with the HTML doctype declaration, which is the first line of an HTML document. It looks like this: <**! DOCTYPE html**>
- Next, create the HTML document structure by adding the <**html**> element. This element contains the entire HTML document and has two parts: the <**head**> section and the <**body**> section.
- Inside the <**head**> section, you can add the title of your document using the <**title**> element. This is the text that appears in the browser's title bar.
- Inside the <**body**> section, you can add the content of your document using various HTML elements such as <**p**> for paragraphs, <**h1**> for headings, <**ul**> and <**li**> for lists, etc.
- Once you've added all the content you want, save your

file with a .html extension.

Here's an example of what your HTML document might look like:

This is just a basic example, but you can add more complex elements, styles, and functionality as you become more comfortable with HTML.

Setting up an HTML Document

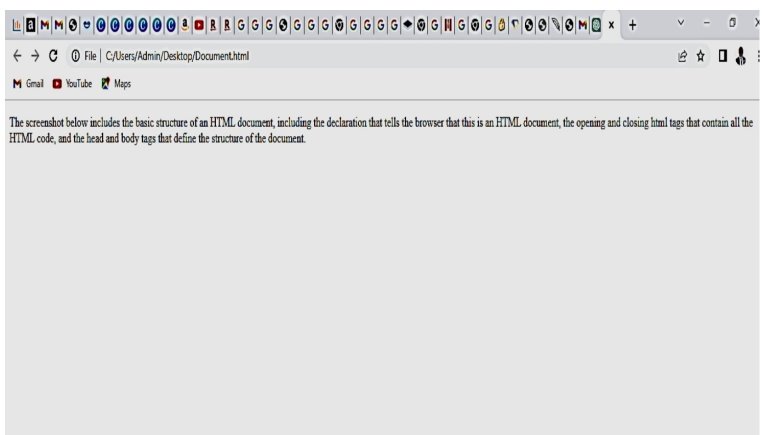
The screenshot below includes the basic structure of an HTML document, including the <**!DOCTYPE html**> declaration that tells the browser that this is an HTML document, the opening and closing **html** tags that contain all the HTML code, and the **head** and **body** tags that define the structure of the document.

To set up an HTML document, you can follow these steps:

- Open a text editor such as Notepad, Sublime Text, or Visual Studio Code.
- Type in the following code to start your HTML document:

```
C: > Users > Admin > Desktop > Document.html > html > body > P
<!DOCTYPE html>
<html>
<head>
    <title>My Document Setup</title>
</head>
<body>
<P>
    The screenshot below includes the basic structure of an HTML document,
    including the <!DOCTYPE html> declaration that tells the browser that this is an HTML document,
     the opening and closing html tags that contain all the HTML code,
     and the head and body tags that define the structure of the document.
</P>
</body>
</html>
```

- Replace the **Page Title** with the title of your page. This title will appear in the browser's tab or title bar.
- Add any content you want to include within the **body** tags. This can include headings, paragraphs, images, links, and other HTML elements.
- Save the file with a **.html** extension, such as **index.html**.
- Open the file in a web browser to view your HTML document. See the screenshot below.

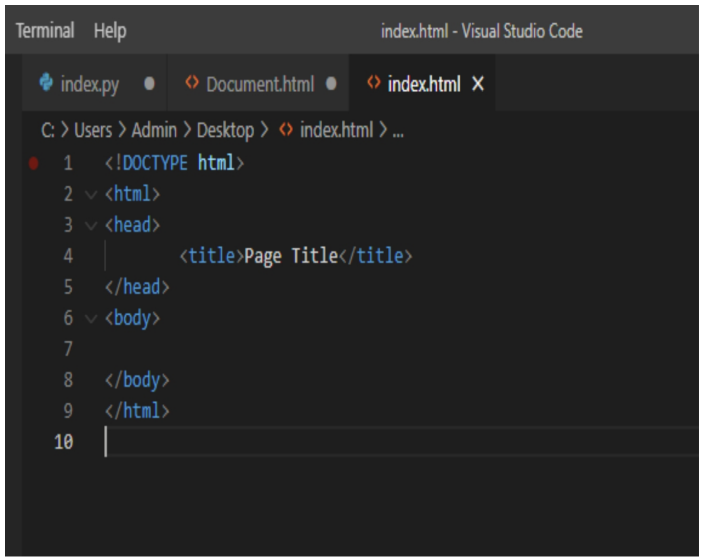

File | C:/Users/Admin/Desktop/Document.html
Gmail
YouTube
Maps
The screenshot below includes the basic structure of an HTML document, including the declaration that tells the browser that this is an HTML document, the opening and closing html tags that contain all the HTML code, and the head and body tags that define the structure of the document.

Adding Content to an HTML Document

To add content to an HTML document, you can use HTML tags to structure and format the content. Here are the basic steps:

- Open a text editor, such as Notepad or Sublime Text.
- Create a new file and save it with a .html extension, for example, "index.html".

- Begin by adding the basic structure of an HTML document using the following tags:

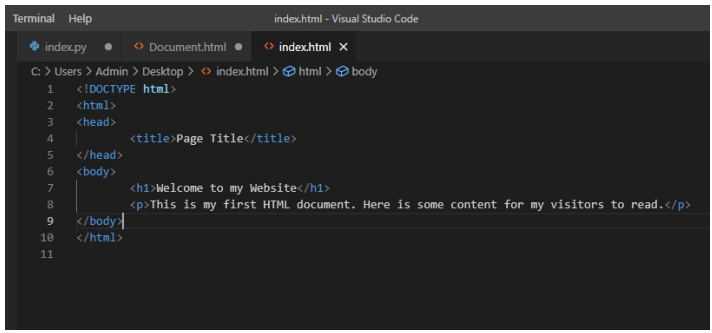


- The <**!DOCTYPE html**> tag specifies the document type and should be included at the beginning of every HTML document.
- The <**html**> tag represents the root element of an HTML document.
- The <**head**> tag contains metadata about the document, such as the title of the page, which is displayed in the browser tab.
- The <**title**> tag specifies the title of the page.
- The <**body**> tag contains the visible content of the document.
- You can add content to the <**body**> tag using various HTML tags such as:

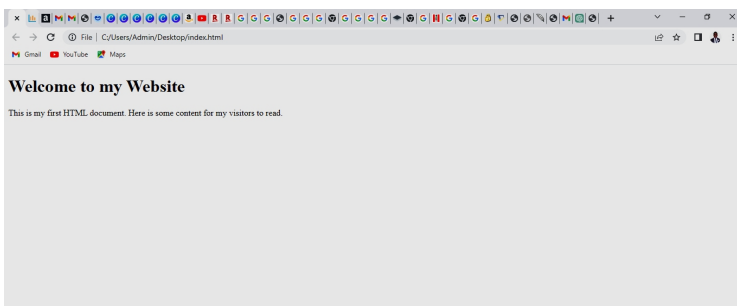
1. <**h1**> to <**h6**> for headings of different sizes
2. <**p**> for paragraphs
3. <**ul**> and <**ol**> for lists
4. <**a**> for links
5. <**img**> for images
6. <**table**> for tables
7. <**div**> and <**span**> for grouping and styling

content

- For example, to add a heading and paragraph to your HTML document, you can use the following code:



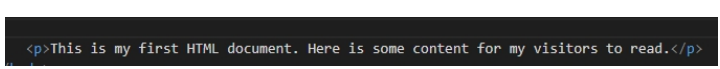
- Save the file and open it in a web browser to see the rendered content.



Understanding HTML Tags, Attributes, and Values

HTML (Hypertext Markup Language) is a markup language used to create web pages. It consists of tags, attributes, and values, which are used to structure and format content on a web page.

HTML Tags: HTML tags are used to define the structure of the content on a web page. They are enclosed in angle brackets (< >) and come in pairs, with a start tag and an end tag. The start tag begins with the name of the tag, and the end tag begins with a forward slash (/) followed by the tag name. For example:



In the above example, the <**p**> tag is the start tag and the </**p**> tag is the end tag. The content "This is a paragraph." is enclosed between these tags and is formatted as a paragraph.

HTML Attributes: HTML attributes are used to provide additional information about HTML elements. They are added to the start tag of an HTML element and consist of a name and a value. The name and value are separated by an equal sign (=), and the value is enclosed in quotation marks. For example:

```
<img src="image.jpg" alt="An image">
```

In the above example, the **src** and **alt** attributes are added to the <**img**> tag. The **src** attribute specifies the URL of the image file, and the **alt** attribute provides alternative text for the image.

HTML Values: HTML values are used to provide a value for an attribute. They are enclosed in quotation marks and can be either a single word or a sentence. For example:

```
<a href="https://www.google.com">Visit Example</a>
```

In the above example, the **href** attribute is used to provide the URL of the page that the link should point to. The value of the **href** attribute is "https://www.google.com". The text "Visit Example" between the <**a**> and </**a**> tags is the content of the link.

Adding Comments to an HTML Document

Comments in an HTML document are used to add notes or explanations that are not visible to the user but can be read by developers who are reviewing the code. To add comments to an HTML document, you can use the following syntax:

```
<!-- This is a comment -->
```

Anything that is written between the <**!--** and **-->** tags will be considered a comment and will not be rendered by the browser.

You can add comments anywhere in the HTML code, including within tags, after tags, or between content.

Here's an example of adding a comment within an HTML tag:

```
<p>This is a paragraph of text.<!-- This is a comment within the paragraph --></p>
```

In this example, the text "This is a comment within the paragraph" is a comment and will not be visible to the user.

CHAPTER TWO

HTML TEXT FORMATTING

In this chapter, we will explore text formatting in HTML. We will cover headings, paragraphs, lists, and links. We will explain how to format text using HTML tags and how to add images to an HTML document. In HTML, text formatting is essential to make your content look presentable and readable.

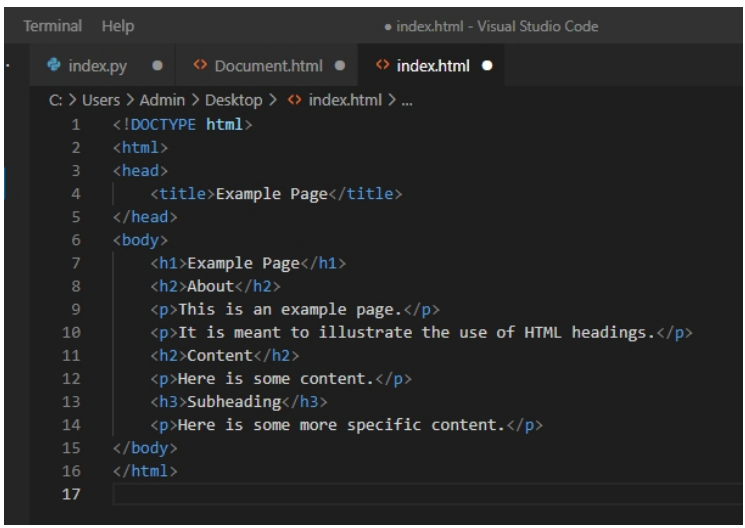
Understanding HTML Headings

HTML headings are used to define the headings or titles of a web

page or a section within a web page. There are six levels of headings in HTML, which are represented by the h1 to h6 tags.

The h1 tag is the highest level of heading and is usually used for the main title of the page. The h2 tag is used for subheadings, and so on, with each subsequent level of heading being used for increasingly smaller and more specific sections of the page.

Here is an example of how HTML headings might be used on a simple web page:



In the example above, the h1 tag is used for the main title of the page, the h2 tag is used for two subheadings, and the h3 tag is used for a subheading under the second subheading. The actual text of the headings and content can be customized as needed to suit the needs of the page.

Understanding HTML Paragraphs

HTML paragraphs are used to structure text into paragraphs, which are blocks of text that typically convey a complete idea or thought.

To create a paragraph in HTML, you can use the element. The element is a container tag that is used to enclose text content to create a paragraph. Here's an example of how to create a paragraph in HTML:

* This is a paragraph. *

In this example, the text "This is a paragraph." is enclosed within the <p> tags, indicating that it is a paragraph.

You can also add additional attributes to the <p> element to provide more information about the paragraph, such as the class or ID:

```
<p class="paragraph">This is a paragraph with a class attribute.</p>
<p id="first-paragraph">This is the first paragraph with an ID attribute.</p>
```

In the first example, the class attribute is set to "paragraph", which can be used to style the paragraph using CSS. In the second example, the ID attribute is set to "first-paragraph", which can be used to create links that point directly to that paragraph. Using paragraphs in HTML can help to organize content on a web page and make it easier to read and understand.

Understanding HTML Lists

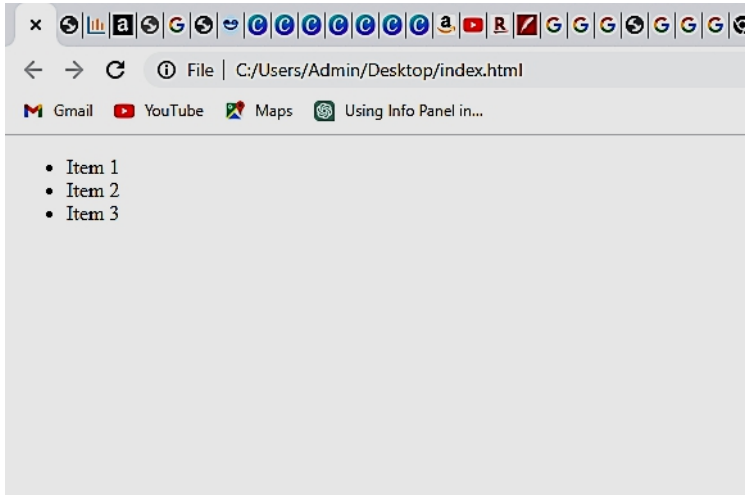
Lists are commonly used in HTML to organize and present information in a structured way. There are two main types of lists in HTML: ordered lists and unordered lists.

Unordered Lists

Unordered lists are used to present a list of items in no particular order. To create an unordered list, you can use the <ul> element. Here's an example:

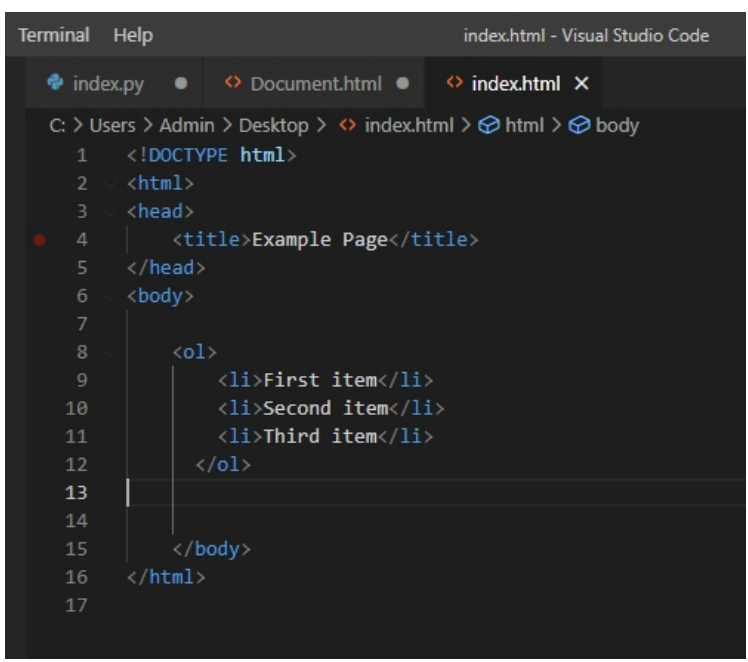
```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
```

In the example, the <ul> element is used to define the unordered list, and each item in the list is defined using the <li> (list item) element. The result will look like this:



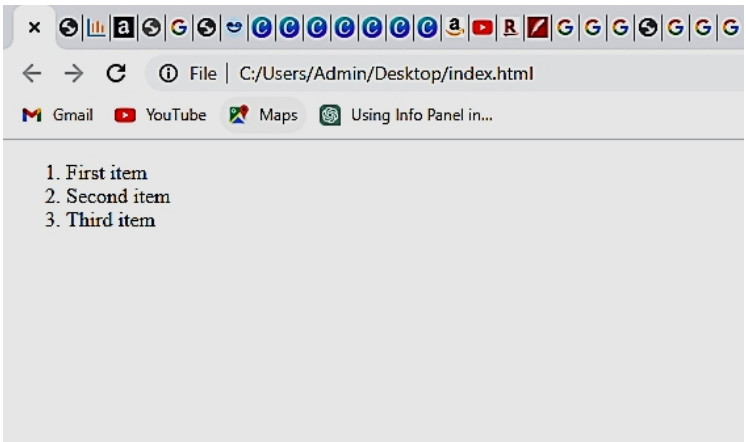
Ordered Lists

Ordered lists are used to present a list of items in a specific order. To create an ordered list, you can use the <ol> element. Here's an example:



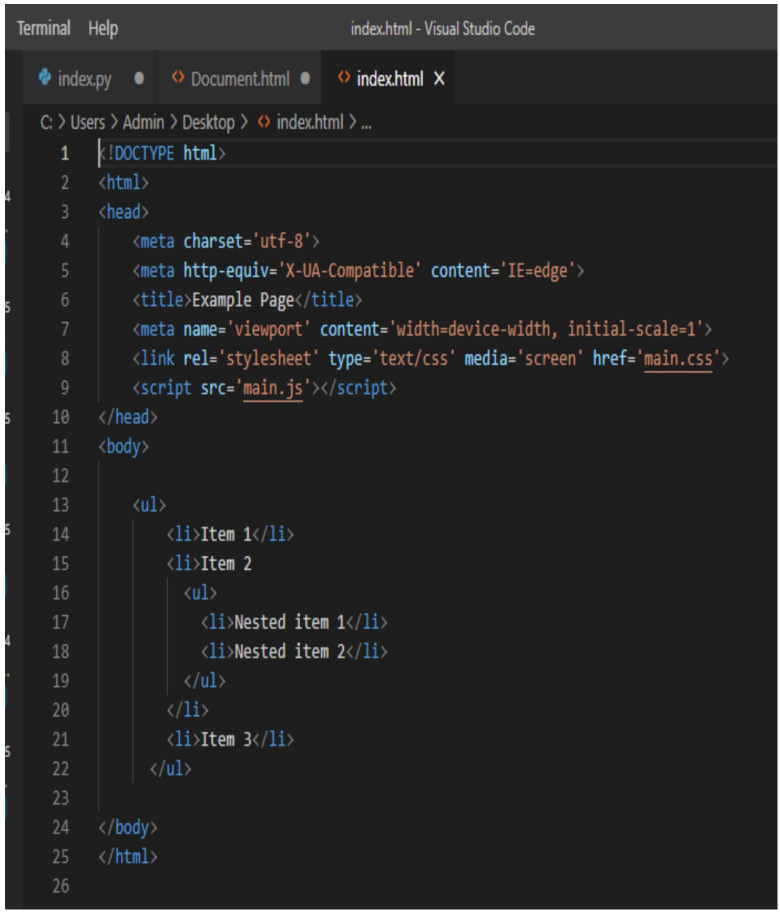
In this example, the <ol> element is used to define the ordered

list, and each item in the list is defined using the <li> element. The result will look like this:

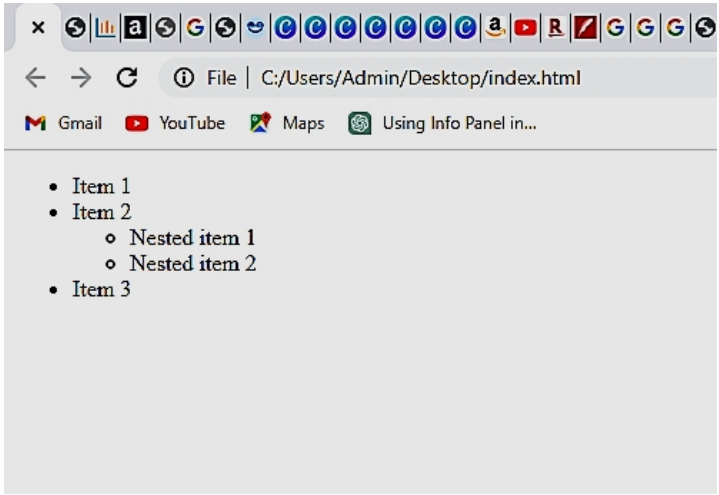


Nested Lists

You can also create nested lists in HTML, which means placing one list inside another list. To do this, you can simply place one list inside another using the appropriate tags. Here's an example:



In this example, a nested unordered list is created by placing a new <ul> element inside the second list item. The result will look like this:



Using lists in HTML can help to organize and present information in a clear and structured way, making it easier for users to understand and navigate.

Understanding HTML links

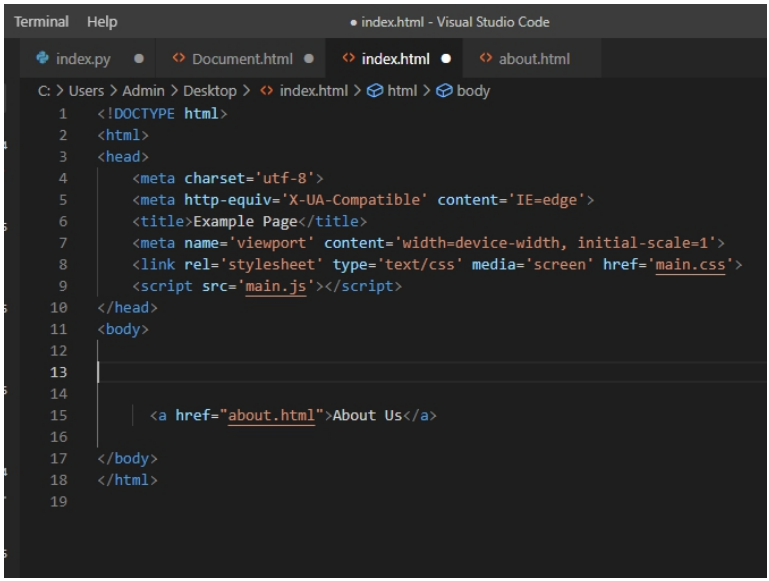
Links are an important aspect of web pages as they allow users to navigate between pages or to other resources on the web. To create a link in HTML, you can use the <a> element, which stands for anchor. Here's an example of how to create a basic link:

```
<a href="http://www.google.com">Link text</a>
```

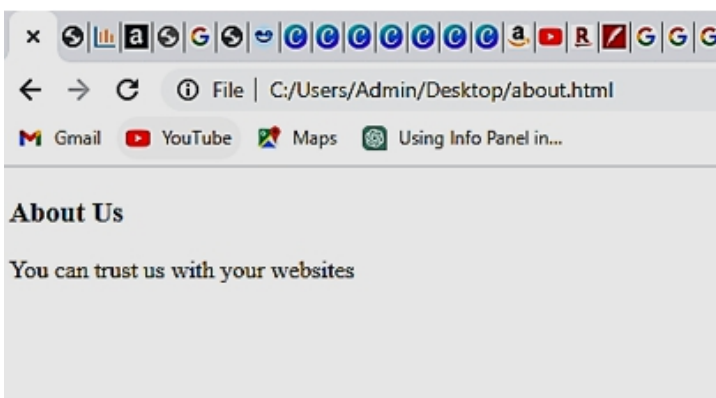
In this example, the href attribute specifies the URL (Uniform Resource Locator) of the destination page or resource, and the link text is the text that the user clicks on to activate the link. When the user clicks on the link text, the browser will navigate to the URL specified in the href attribute.

Relative URLs

In addition to absolute URLs, you can also use relative URLs to link to pages or resources within the same website. Relative URLs specify the path to the page or resource relative to the current page. Here's an example:

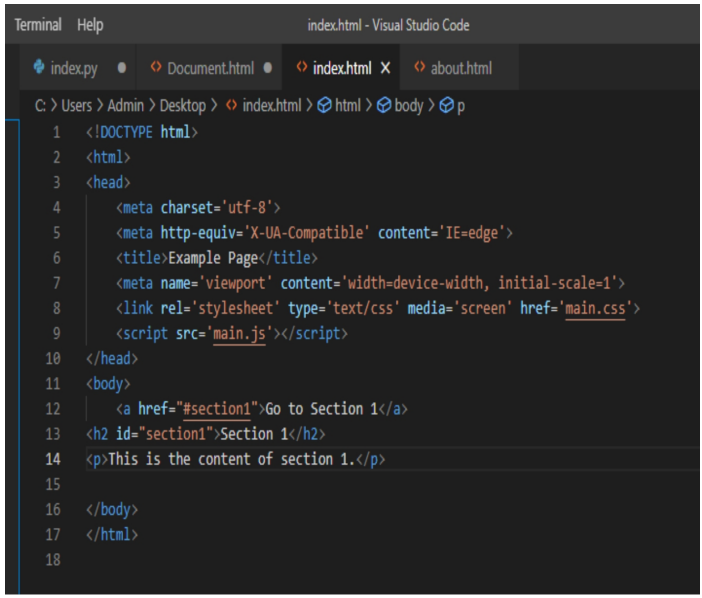


In this example, the link will navigate to the "about.html" page in the same directory as the current page.

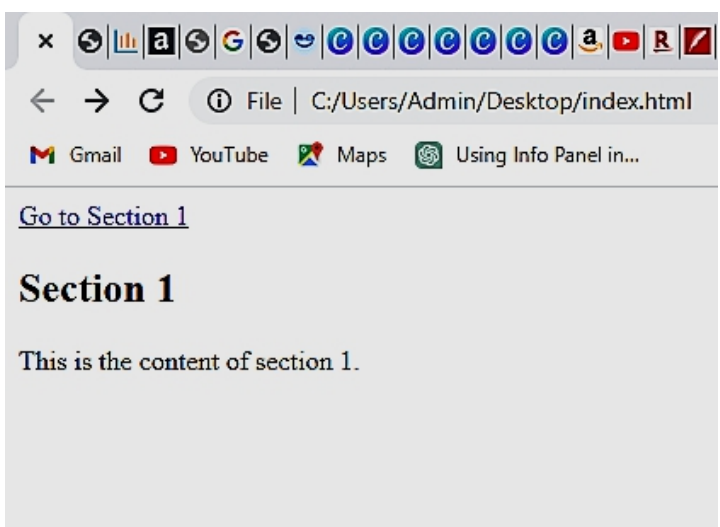


Linking to Specific Parts of a Page

You can also create links that navigate to specific parts of a page by using the id attribute. Here's an example:



In this example, the link text "Go to Section 1" will navigate to the section of the page with the id attribute set to "section1". When the link is clicked, the page will scroll to the section and highlight it. Using links in HTML is an important part of creating a user-friendly website. By using descriptive link text and providing clear and relevant destinations, you can help users navigate your website more easily. See the result below:



How to Format Text Using HTML Tags

HTML tags are used to format and structure text on a web page. Here are some common HTML tags and how they are used to format text:

- **Headings**: Headings are used to organize the content of the page and provide a visual hierarchy. There are six levels of headings, from h1 (the most important) to h6 (the least important). To use a heading tag, simply wrap your text in the appropriate tag.
- **Paragraphs**: Paragraphs are used to separate blocks of text. To create a paragraph, simply wrap your text in the <p> tag.
- **Bold and Italic**: You can emphasize text using the tag for bold text and the tag for italic text.
- **Lists**: There are two types of lists in HTML: ordered lists (<ol>) and unordered lists (<ul>). To create a list, wrap your list items in the appropriate tags:
- **Links**: To create a hyperlink, use the <a> tag and include the URL in the href attribute.

These are just a few of the most commonly used HTML tags for formatting text. There are many more tags available for formatting text, images, tables, and other elements on a web page.

How to Add Images to an HTML Document

To add an image to an HTML document, you can use the <img> tag. Here's how:

- Save the image file to your computer in a location that you can easily find, such as your desktop.
- Open your HTML file in a text editor or an HTML editor.

In the HTML file, add a <img> tag where you want the image to appear. The <img> tag has two required attributes:

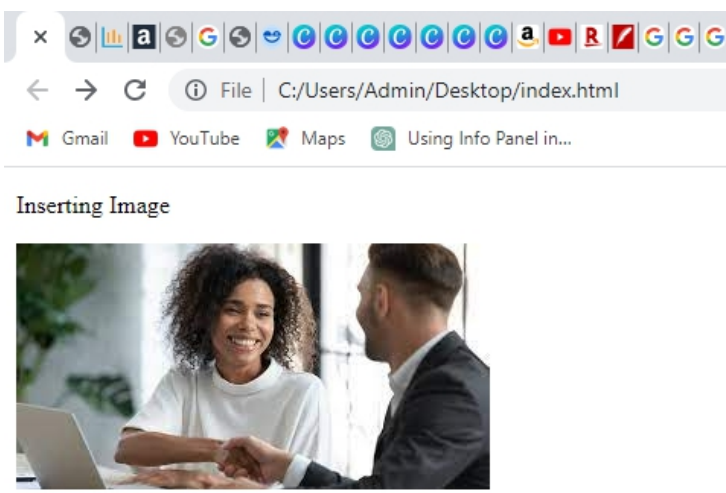
- src: This specifies the URL of the image file.
- alt: This provides alternative text for the image, which

is displayed if the image can't be loaded.

Here's an example of a <img> tag:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Example Page</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' type='text/css' media='screen' href='main.css'>
    <script src='main.js'></script>
</head>
<body>
    <p>Inserting Image</p>
    <img src="img1.jpg" alt="Career Guide For You!">
</body>
</html>
```

In this example, replace the image path with the actual file path of your image on your computer. Also, add a short description of the image. Save the HTML file and open it in your web browser to see the image as follows.



Note that the src attribute can use either a relative or absolute file path to specify the location of the image file. If the image file is in the same directory as the HTML file, you can use a relative file path

like this:

```
<img src="image.jpg" alt="Career Guide for You!">
```

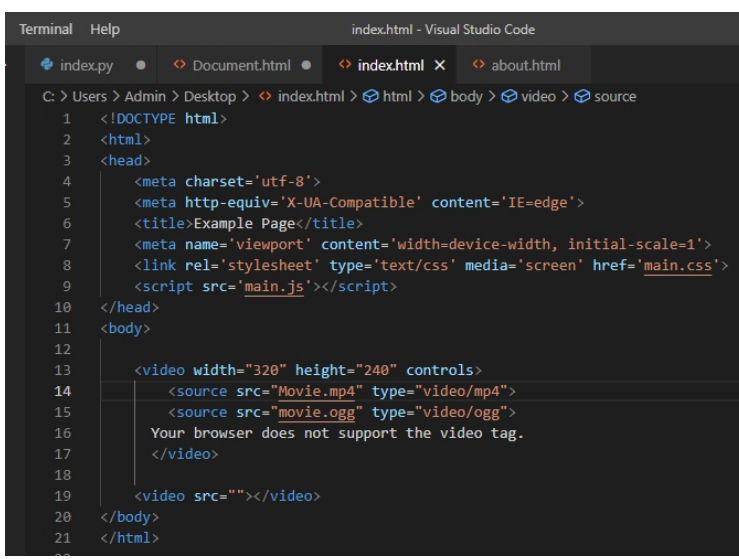
If the image file is in a different directory, you'll need to specify the path to that directory, like this:

```
<img src="images/image.jpg" alt="Career Guide for You!">
```

In this example, the image file is located in a subdirectory called images within the directory containing the HTML file.

How to Add Videos to an HTML Document

To add videos to an HTML5 document, you can use the <**video**> element. Here's an example of how to use the <**video**> element:

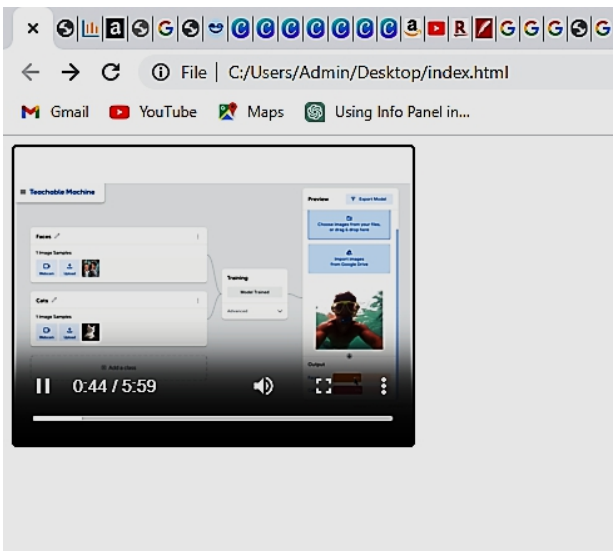


In this example, the <**video**> element creates a video player on the web page, with the **controls** attribute indicating that the player should include basic playback controls such as play, pause, and

volume.

The <**source**> element within the <**video**> element specifies the location of the video file, along with its MIME type. In this example, there are two <**source**> elements, one for the MP4 format and one for the WebM format. The browser will use the first format it can support. You can include additional <**source**> elements to support other video formats.

The text within the <**video**> element (in this case, "Your browser does not support the video tag.") is displayed if the user's browser does not support the <**video**> element or any of the specified video formats. See the output of the above screenshot below:



Here are some additional attributes you can use with the <**video**> element:

- **width**: Specifies the width of the video player, in pixels.
- **height**: Specifies the height of the video player, in pixels.
- **autoplay**: Specifies that the video should start playing automatically when the web page loads.
- **loop**: Specifies that the video should play in a continuous loop.
- **muted**: Specifies that the video should play without

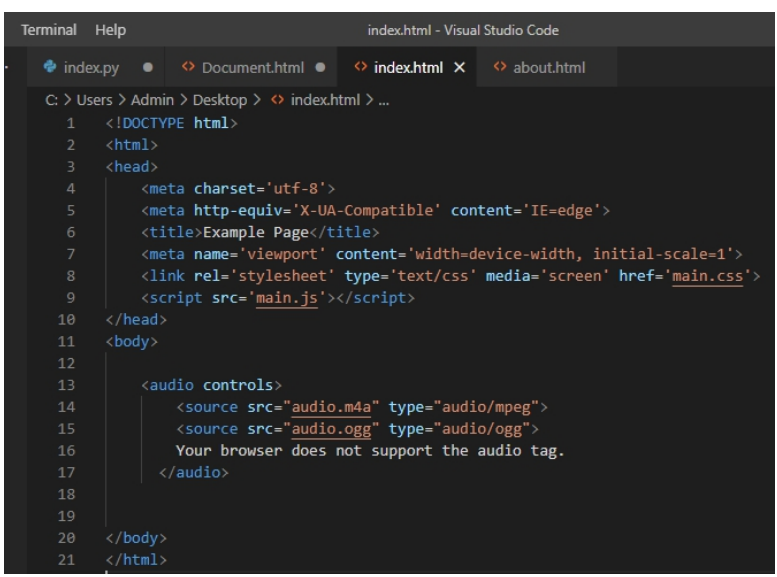
sound.

Keep in mind that video files can be large, so it's important to optimize them for web playback to ensure that they load quickly and don't use too much bandwidth. This can include compressing the video file and using the appropriate video format for your needs.

Using Audio in HTML5

To embed audio content in an HTML or XHTML document, HTML5 supports the audio> tag. The audio tag in the current HTML5 draft specification does not list the audio formats that browsers should support. Ogg, mp3, and wav are the audio formats that are most frequently used.

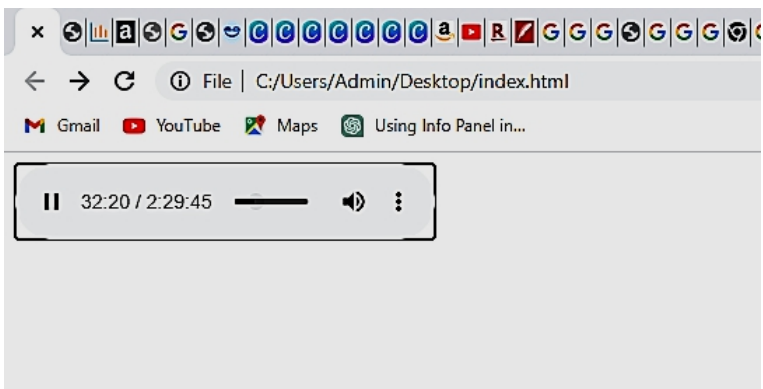
Before HTML5, audio files could be introduced to a page by using the bgsound> tag to integrate background sound. The user was unable to silence the sound, which was played while the user was viewing the page. With HTML5, there is no need to link third-party plugins because audio files may be included using the audio> tag. HTML or Javascript can be used to control the audio element, and CSS can be used to style it. To use audio in HTML5, you can use the <**audio**> element. Here's an example of how to use the <**audio**> element:



In this example, the <**audio**> element creates an audio player on the web page, with the **controls** attribute indicating that the player should include basic playback controls such as play, pause, and volume.

The <**source**> element within the <**audio**> element specifies the location of the audio file, along with its MIME type. In this example, there are two <**source**> elements, one for the MP3 format and one for the Ogg Vorbis format. The browser will use the first format it can support. You can include additional <**source**> elements to support other audio formats.

The text within the <**audio**> element (in this case, "Your browser does not support the audio tag.") is displayed if the user's browser does not support the <**audio**> element or any of the specified audio formats. See the result of the code above below.



Here are some additional attributes you can use with the <**audio**> element:

- **autoplay**: Specifies that the audio should start playing automatically when the web page loads.
- **loop**: Specifies that the audio should play in a continuous loop.
- **muted**: Specifies that the audio should play without sound.

You can also use JavaScript to control the playback of the audio, for example, to add additional controls or to change the playback speed.

CHAPTER THREE

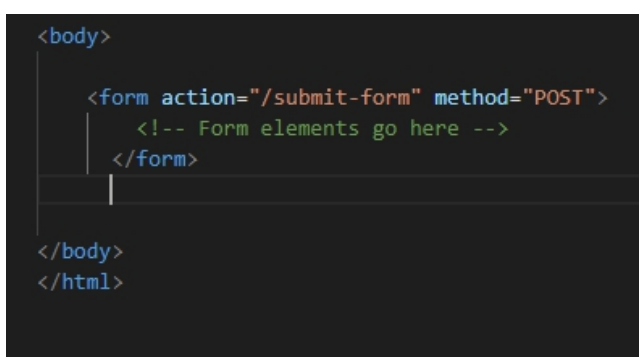
UNDERSTANDING FORMS AND INPUT

Understanding forms and input is an essential part of web development. Forms allow users to submit information to a website, whether it's creating an account, signing up for a newsletter, or making a purchase. In this chapter, we will explore the basics of creating forms and the different types of form elements, such as text input fields, radio buttons, checkboxes, and select menus. We will also discuss how to style form elements to create a cohesive and visually appealing user interface. Understanding forms and input is crucial for creating effective and user-friendly web applications, so let's dive in!

Creating Forms

Creating forms involves using several elements and attributes to allow users to input data or information that can be submitted to a server for processing. Here are the basic steps to create a form in HTML5:

1. Use the <**form**> element to create a form on the web page. The **action** attribute specifies the URL where the form data will be submitted, and the **method** attribute specifies the HTTP method to use when submitting the form data (either **GET** or **POST**).

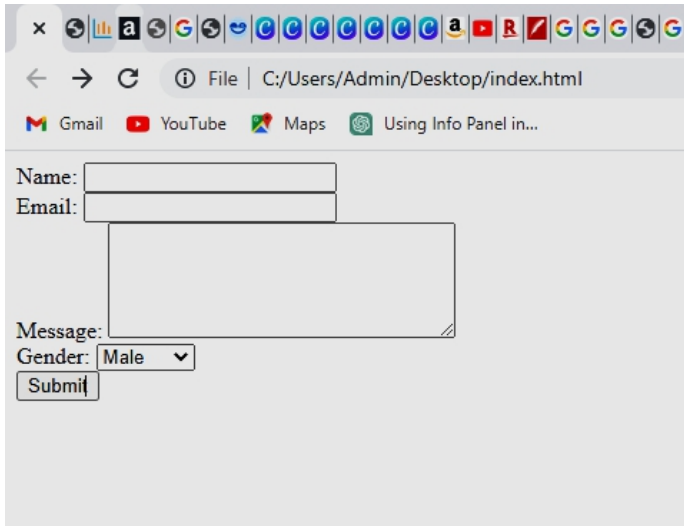


1. Use form elements such as <**input**>, <**label**>, <**textarea**>, <**select**>, and <**button**> to create form fields, labels, and buttons.

Example:

```
<body>

    <form action="/submit-form" method="POST">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <br>

        <label for="message">Message:</label>
        <textarea id="message" name="message" rows="5" cols="30"></textarea>
        <br>

        <label for="gender">Gender:</label>
        <select id="gender" name="gender">
          <option value="male">Male</option>
          <option value="female">Female</option>
        </select>
        <br>

        <button type="submit">Submit</button>
      </form>

</body>
</html>
```

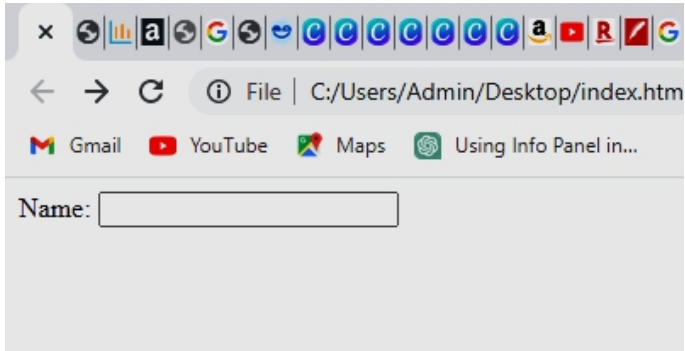
In this example, the form includes several form elements such as text input fields for name and email, a text area for the message, a dropdown list for gender, and a submit button. Each form element has its unique attributes, such as **type**, **id**, **name**, and **value**. See the output below:



1. Use the **for** attribute in the <**label**> element to link the label to its corresponding form element using the **id** attribute. This improves accessibility and user experience by making it easier to select the correct form element.

```
    <form action="/submit-form" method="POST">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <br>
        <!-- More form elements -->
    </form>


</body>
</html>
```

See the output below.



1. Use the **required** attribute on the form element to make it mandatory for the user to enter a value in the form field before submitting the form.

Example:

```
<body>

    <form action="/submit-form" method="POST">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <br>
        <!-- More form elements -->
    </form>

</body>
</html>
```

1. Use the **type** attribute in the <**input**> element to specify the type of input field you want to use, such as text, email, password, radio buttons, checkboxes, etc.

Example:

```
    <form action="/submit-form" method="POST">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <br>
        <!-- More form elements -->
    </form>


</body>
</html>
```

Text Inputs and Labels

In HTML, text inputs and labels are used to create user input fields on a web page. Text inputs allow users to enter text, while labels are used to describe what type of information should be entered into the input field.

To create a text input field, you can use the <**input**> tag with the **type**="**text**" attribute. For example:

```
<body>

    <label for="name">Name:</label>
<input type="text" id="name" name="name">


</body>
</html>
```

In the above code, the **label** tag is used to describe what information should be entered into the input field. The **for** attribute in the label tag is used to associate the label with the input field using the **id** attribute. The **id** attribute is used to uniquely identify the input field. The **name** attribute is used to give the input field a name that can be used when the form is submitted.

To create a label, you can use the <**label**> tag. The **for** attribute in the label tag should be set to the **id** attribute of the corresponding input field. This helps screen readers and other accessibility tools associate the label with the input field.

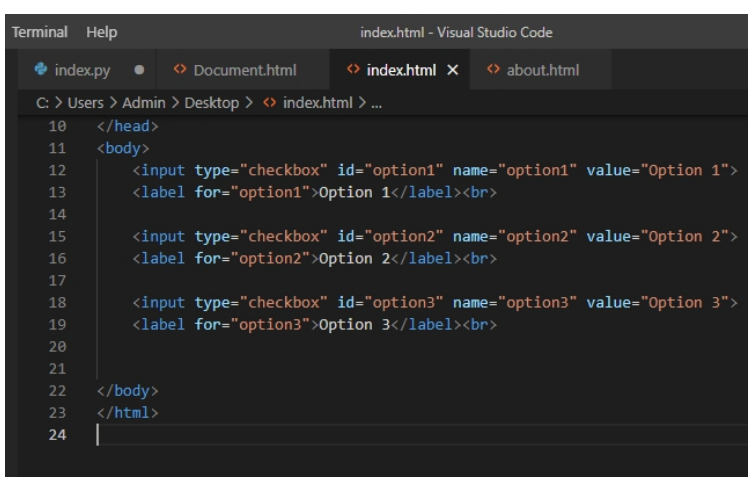
```
<body>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email">

</body>
</html>
```

In the above code, the **type** attribute is set to "email" to ensure that the user enters a valid email address. You can also use other input types, such as "password", "number", "date", and more, to create input fields that accept different types of data.

Checkboxes and Radio Buttons

Checkboxes and radio buttons are two types of input fields that are commonly used in web forms to allow users to make selections from a list of options.

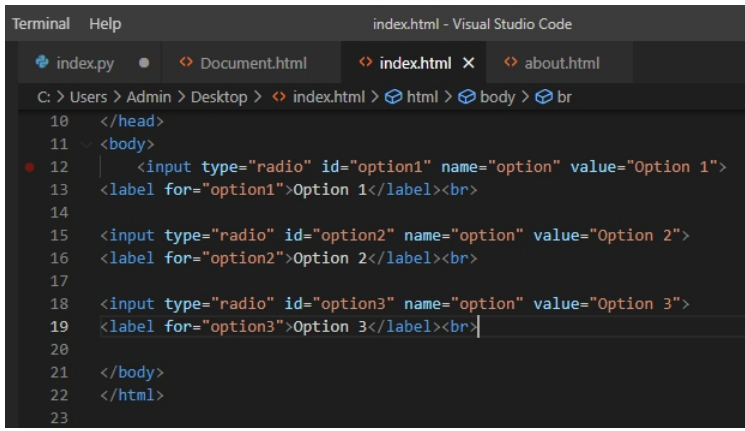
A checkbox allows the user to select one or more options from a list of choices. To create a checkbox in HTML, you can use the <**input**> tag with the **type**="**checkbox**" attribute.



In the above code, each checkbox has a unique **id** and **name** attribute, which allows the server to process the user's selections.

The **value** attribute is used to define the value associated with each checkbox when the form is submitted.

A radio button, on the other hand, allows the user to select only one option from a list of choices. To create a radio button in HTML, you can use the <**input**> tag with the **type**="**radio**" attribute.



In the above code, each radio button has the same **name** attribute, which ensures that only one option can be selected at a time. When the form is submitted, the value of the selected radio button is sent to the server.

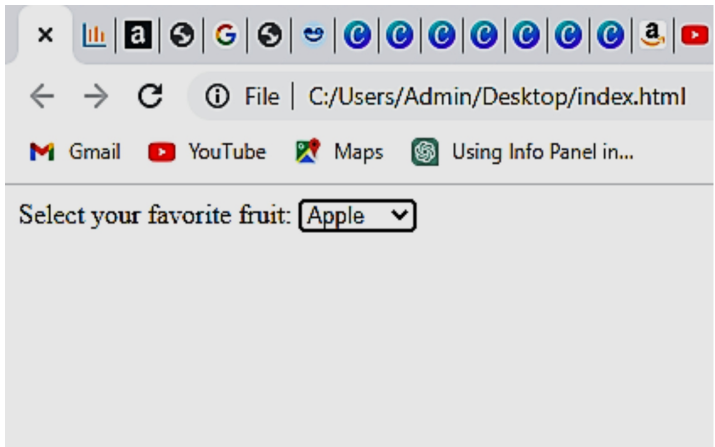
Select Menus and Option Groups

Select menus and option groups are two more types of input fields that are commonly used in web forms to allow users to make selections from a list of options.

A select menu is a drop-down list of options that the user can choose from. To create a select menu in HTML, you can use the <**select**> tag with one or more <**option**> tags inside.

```
<body>
    <label for="fruit">Select your favorite fruit:</label>
    <select id="fruit" name="fruit">
      <option value="apple">Apple</option>
      <option value="banana">Banana</option>
      <option value="orange">Orange</option>
      <option value="mango">Mango</option>
    </select>

</body>
</html>
```

See the output below.

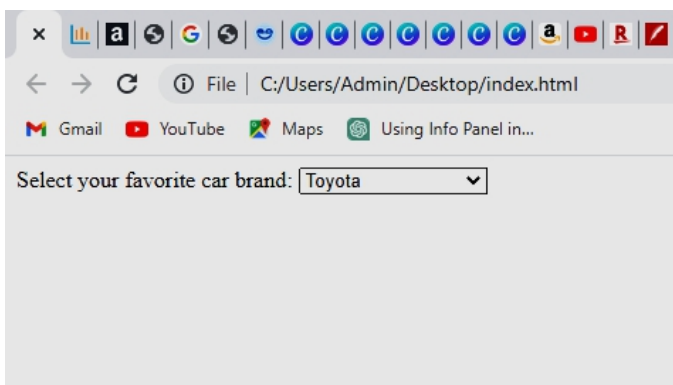


In the above code, the **id** and **name** attributes are used to uniquely identify the select menu. Each <**option**> tag has a **value** attribute that defines the value associated with that option when the form is submitted.

Option groups are used to group related options together in a select menu. To create an option group in HTML, you can use the <**optgroup**> tag with one or more <**option**> tags inside.

```
<body>
    <label for="cars">Select your favorite car brand:</label>
<select id="cars" name="cars">
  <optgroup label="Japanese Cars">
    <option value="toyota">Toyota</option>
    <option value="honda">Honda</option>
    <option value="nissan">Nissan</option>
  </optgroup>
  <optgroup label="European Cars">
    <option value="bmw">BMW</option>
    <option value="mercedes">Mercedes-Benz</option>
    <option value="audi">Audi</option>
  </optgroup>
</select>

</body>
</html>
```

See the output below.



In the above code, the <**optgroup**> tag is used to group the options into two categories: Japanese Cars and European Cars. The **label** attribute is used to define the name of each option group.

Text Areas and Buttons

Text areas and buttons are two additional types of input fields that are commonly used in web forms to allow users to provide more detailed information or to submit the form data.

A text area is a multi-line input field that allows the user to enter a larger amount of text. To create a text area in HTML, you can use the <**textarea**> tag with the **cols** and **rows** attributes to define

the size of the input field.

```
<body>
    <label for="message">Enter your message:</label>
<textarea id="message" name="message" rows="5" cols="30"></textarea>


</body>
</html>
```

In the above code, the **id** and **name** attributes are used to uniquely identify the text area, and the **rows** and **cols** attributes are used to define the number of rows and columns in the text area.

A button is an input field that the user can click to submit the form or perform some other action. There are three types of buttons in HTML: submit, reset, and button. The submit button is used to submit the form data, the reset button is used to reset the form to its initial state, and the button type is used for custom actions.

```
<body>
    <button type="submit">Submit</button>
<button type="reset">Reset</button>
<button type="button">Click me</button>

</body>
</html>
```

In the above code, each button has a **type** attribute that specifies the type of button. The text inside the button tags is the label that appears on the button.

Understanding Form Validation

Form validation is the process of checking the user input in a form to ensure that it meets certain requirements or constraints. Validation is important because it helps to ensure the accuracy and consistency of the data collected through web forms.

HTML5 provides built-in form validation features that allow you to specify validation requirements for input fields using attributes. The validation is done automatically in modern web browsers, and the

user will be prompted with error messages if the input does not meet the specified requirements.

Some of the common attributes that are used for form validation in HTML5 include:

- **required**: Specifies that the input field must be filled out before the form can be submitted.
- **pattern**: Specifies a regular expression pattern that the input must match.
- **min** and **max**: Specifies the minimum and maximum allowed values for a numeric input field.
- **minlength** and **maxlength**: Specifies the minimum and maximum lengths of a text input field.
- **type**: Specifies the type of input, such as email, url, number, or date, and the browser will validate the input accordingly.

Here is an example of how to use some of these validation attributes:

```
<body>
    <form>
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required minlength="5">

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>

        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required minlength="8">

        <label for="age">Age:</label>
        <input type="number" id="age" name="age" min="18" max="99">

        <button type="submit">Submit</button>
    </form>

</body>
</html>
```

In the above code, the **required** attribute is used to ensure that the username, email, and password fields are filled out before the form can be submitted. The **minlength** attribute is used to specify that the username and password fields must be at least 5 and 8 characters long, respectively. The **type** attribute is used to specify the email and number input fields and to enable the browser to validate them accordingly.

CHAPTER FOUR

UNDERSTANDING TABLES AND LISTS

Tables and lists are important HTML elements used to organize and present information in a structured and readable format on web pages. Tables are commonly used to display data in rows and columns, while lists are used to display content in a bulleted or numbered format.

In this chapter, we will explore how to create and style tables and lists in HTML. We will also cover some of the advanced features of tables, such as colspan and rowspan, and show how to create nested lists. Additionally, we will discuss some best practices for designing and formatting tables and lists to ensure they are accessible and easy to read for all users.

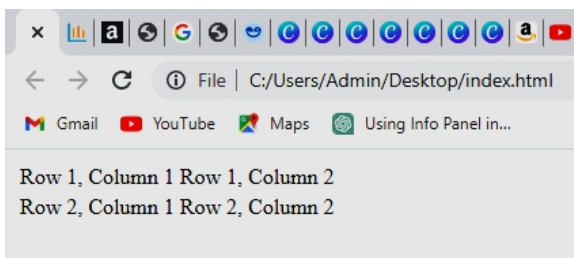
By the end of this chapter, you will have a good understanding of how to use tables and lists to organize and present content on your web pages. You will also be able to apply various styling techniques to enhance the visual appearance of your tables and lists.

Creating Tables

To create a table in HTML, you will need to use the <**table**> element. Within the table element, you can use other elements to create rows and cells. Here is a basic example of how to create a simple table with two rows and two columns:

```
<body>
    <table>
        <tr>
            <td>Row 1, Column 1</td>
            <td>Row 1, Column 2</td>
        </tr>
        <tr>
            <td>Row 2, Column 1</td>
            <td>Row 2, Column 2</td>
        </tr>
      </table>

</body>
</html>
```

See the result below.

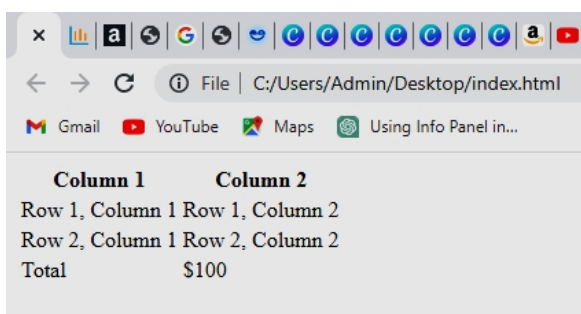


In the above code, we have created a table with two rows and two columns. The <**tr**> element represents a table row, and the <**td**> element represents a table cell. The text within each <**td**> element will be displayed in the corresponding cell in the table.

By default, the table cells will be separated by borders, but you can use CSS to change the border style, color, and thickness. You can also add other elements such as <**thead**>, <**tbody**>, and <**tfoot**> to further structure your table and add headers or footers.

```
<body>
    <table>
        <thead>
          <tr>
            <th>Column 1</th>
            <th>Column 2</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>Row 1, Column 1</td>
            <td>Row 1, Column 2</td>
          </tr>
          <tr>
            <td>Row 2, Column 1</td>
            <td>Row 2, Column 2</td>
          </tr>
        </tbody>
        <tfoot>
          <tr>
            <td>Total</td>
            <td>$100</td>
          </tr>
        </tfoot>
      </table>

</body>
</html>
```

See the result of the code below.



In this example, we have added an <**thead**> element to create a table header with two columns, and a <**tfoot**> element to create a table footer with a summary of the data. The <**tbody**> element contains the actual data rows of the table.

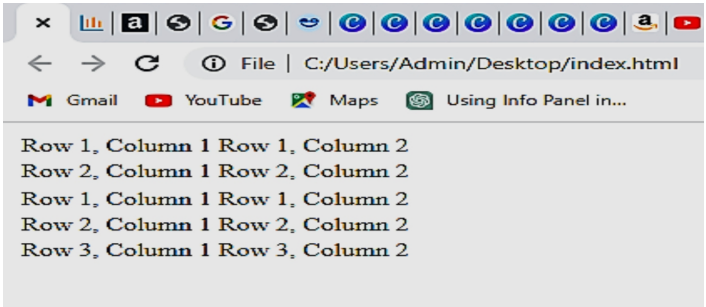
Tables can also be made more complex with the use of attributes such as **colspan** and **rowspan**, which allow cells to span multiple columns or rows. Additionally, you can use CSS to style the table with borders, background colors, and font styles.

Adding Rows and Columns

To add rows and columns to an existing table in HTML, you can use the <**tr**> and <**td**> elements within the table. Here's an example of how to add a row to an existing table:

```
<body>
    <table>
        <tr>
          <td>Row 1, Column 1</td>
          <td>Row 1, Column 2</td>
        </tr>
        <tr>
          <td>Row 2, Column 1</td>
          <td>Row 2, Column 2</td>
        </tr>
      </table>

      <!-- Adding a row -->
      <table>
        <tr>
          <td>Row 1, Column 1</td>
          <td>Row 1, Column 2</td>
        </tr>
        <tr>
          <td>Row 2, Column 1</td>
          <td>Row 2, Column 2</td>
        </tr>
        <tr>
          <td>Row 3, Column 1</td>
          <td>Row 3, Column 2</td>
        </tr>
      </table>
</body>
</html>
```

See the result of the code above below.

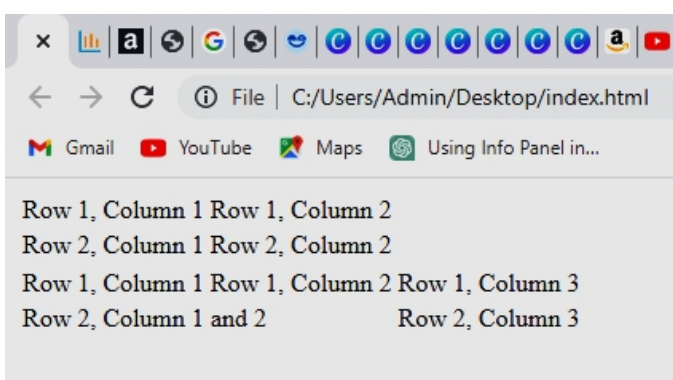

File | C:/Users/Admin/Desktop/index.html
Gmail
YouTube
Maps
Using Info Panel in...
Row 1, Column 1 Row 1, Column 2
Row 2, Column 1 Row 2, Column 2
Row 1, Column 1 Row 1, Column 2
Row 2, Column 1 Row 2, Column 2
Row 3, Column 1 Row 3, Column 2

In this example, we have added a new row to the existing table by inserting a new <**tr**> element with two <**td**> elements containing the text for the new row.

To add a column to an existing table, you can use the **colspan** attribute to make a cell span multiple columns:

```
<body>
    <table>
        <tr>
          <td>Row 1, Column 1</td>
          <td>Row 1, Column 2</td>
        </tr>
        <tr>
          <td>Row 2, Column 1</td>
          <td>Row 2, Column 2</td>
        </tr>
      </table>

      <!-- Adding a column -->
      <table>
        <tr>
          <td>Row 1, Column 1</td>
          <td>Row 1, Column 2</td>
          <td>Row 1, Column 3</td>
        </tr>
        <tr>
          <td colspan="2">Row 2, Column 1 and 2</td>
          <td>Row 2, Column 3</td>
        </tr>
      </table>

</body>
</html>
```

See the result of the code above below.


File | C:/Users/Admin/Desktop/index.html
Gmail
YouTube
Maps
Using Info Panel in...
Row 1, Column 1 Row 1, Column 2
Row 2, Column 1 Row 2, Column 2
Row 1, Column 1 Row 1, Column 2 Row 1, Column 3
Row 2, Column 1 and 2
Row 2, Column 3

In this example, we have added a new column to the existing table by adding a new <**td**> element to each row and then used the

colspan attribute to make the second cell in the second-row span two columns. This creates a table with three columns, where the second cell in the second row spans two columns.

Styling Tables with CSS

You can style tables in HTML using CSS to change their appearance, layout, and behavior. Here are some examples of how to style tables with CSS:

- Changing the background color of the table:

```
table {
  background-color: #f2f2f2;
}
```

- Changing the font size and color of the table headers:

```
th {
  font-size: 18px;
  color: blue;
}
```

- Setting a fixed width for the table and its columns:

```
    table {
  width: 100%;
  table-layout: fixed;
}

th, td {
  width: 25%;
}
```

- Adding borders and spacing to the table cells:

```
    table {
  border-collapse: collapse;
  border-spacing: 0;
}

td {
  border: 1px solid black;
  padding: 10px;
}
```

- Styling alternating rows with different background colors:

```
    tr:nth-child(even) {
  background-color: #f2f2f2;
}

tr:nth-child(odd) {
  background-color: #fff;
}
```

Creating Ordered and Unordered Lists

In HTML, you can create both ordered lists and unordered lists using the <**ol**> and <**ul**> tags, respectively. Here's an example of how to create an unordered list:

```
<body>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>

</body>
</html>
```

And here's an example of how to create an ordered list:

```
<body>
    <ol>
        <li>First item</li>
        <li>Second item</li>
        <li>Third item</li>
    </ol>

</body>
</html>
```

You can also nest lists inside other lists to create more complex structures. Here's an example of a nested list:

```
<body>

    <ul>
        <li>Item 1</li>
        <li>Item 2
          <ul>
            <li>Subitem 1</li>
            <li>Subitem 2</li>
          </ul>
        </li>
        <li>Item 3</li>
      </ul>

</body>
</html>
```

When creating lists, you can also use the **type** attribute to specify the type of list marker for ordered lists. For example, you can use the **type** attribute to create a Roman numeral list like this:

```
<body>

    <ol type="I">
        <li>First item</li>
        <li>Second item</li>
        <li>Third item</li>
      </ol>

</body>
</html>
```

You can also use CSS to style lists, for example by changing the color or size of the markers, or by adding background colors or borders to the list items.

CHAPTER FIVE

ADVANCED HTML5 FEATURES

Advanced HTML5 features are powerful tools that can be used to

create complex and dynamic web applications. These features include things like canvas, web sockets, web storage, geolocation, and more. In this chapter, we'll take a closer look at some of these advanced features and how they can be used to enhance the functionality and interactivity of your web pages.

Canvas, for example, is a feature that allows you to create dynamic graphics and animations directly within your web page. Web sockets provide a way for real-time communication between a web browser and a server, enabling features like chat applications and online games. Web storage allows you to store data on the client-side, improving performance and reducing the amount of data sent back and forth between the server and client.

By understanding and utilizing these advanced HTML5 features, you can create rich and engaging web applications that provide a better user experience and drive greater engagement and interaction with your content. In this chapter, we'll explore these features in more detail and show you how to use them effectively in your web development projects.

Canvas and SVG graphics

Canvas and SVG are two popular ways of creating graphics in HTML5. Canvas is a powerful feature that allows you to draw and manipulate graphics directly within your web page using JavaScript. With Canvas, you can create dynamic animations, games, interactive charts, and more. It works by providing a bitmap canvas on which you can draw shapes, images, and other elements using JavaScript. The canvas can then be animated and manipulated in real-time, making it a powerful tool for creating dynamic and interactive web applications.

SVG (Scalable Vector Graphics) is a markup language for describing two-dimensional vector graphics. Unlike Canvas, SVG is not a bitmap-based technology and allows you to create images that are infinitely scalable without loss of quality. With SVG, you can create complex graphics, such as charts, diagrams, logos, and even animations, by using XML-based code. SVG images are also interactive, allowing you to add links, buttons, and other elements to create engaging and interactive user experiences.

While both Canvas and SVG can be used to create graphics in HTML5, they are fundamentally different technologies with different use cases. Canvas is best suited for creating dynamic and interactive graphics that require real-time manipulation, such as games and animations. On the other hand, SVG is ideal for creating high-quality, scalable vector graphics that can be used in a variety of contexts, such as logos, charts, and diagrams.

How to Apply Canva and SVG Graphics

To apply Canvas graphics in HTML5, you will need to create a Canvas element in your HTML document, and then use JavaScript to draw shapes, images, and other elements on the canvas. Here's a basic example of how to create a Canvas element:

```
<canvas id="myCanvas" width="500" height="500"></canvas>
```

In the example above, we create a Canvas element with an ID of "myCanvas" and set its width and height to 500 pixels. To draw on the canvas, you can use JavaScript to access the canvas context and use drawing methods to create shapes, lines, and other elements. Here's an example of how to draw a rectangle on the canvas:

```
    var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
ctx.fillStyle = "red";
ctx.fillRect(50, 50, 100, 100);
```

In this example, we use JavaScript to get the Canvas element by its ID and then access its context using the **getContext()** method. We then set the fill style to red and use the **fillRect()** method to draw a rectangle on the canvas with a position of (50, 50) and a width and height of 100 pixels.

To apply SVG graphics in HTML5, you can create an SVG element in your HTML document and then use SVG markup language to define shapes, lines, and other elements. Here's a basic example of how to create an SVG element:

```
    <svg width="500" height="500">
  <rect x="50" y="50" width="100" height="100" fill="red"/>
</svg>
```

In this example, we create an SVG element with a width and height of 500 pixels. We then use the SVG **rect** element to draw a rectangle with a position of (50, 50) and a width and height of 100 pixels, with the fill color set to red.

Both Canvas and SVG graphics can be styled with CSS to further customize their appearance, and both can be animated using JavaScript to create dynamic and interactive user experiences.

Understanding Web Storage

Web storage is a feature of modern web browsers that allows web developers to store key-value pairs of data in a user's browser. This data can be stored either temporarily (session storage) or permanently (local storage). Web storage provides a way for web developers to create more responsive and interactive web applications that can store data on the client-side.

Session storage stores data for a specific browsing session and the data is cleared when the user closes the browser window. This type of storage is useful for storing temporary data that is needed for the current session, such as a user's preferences for a particular web application.

Local storage, on the other hand, stores data permanently in the browser until it is cleared by the user or the web application. This type of storage is useful for storing data that needs to persist across multiple browsing sessions, such as a user's login information or settings for a web application.

To use web storage in HTML5, you can use the **localStorage** and **sessionStorage** objects in JavaScript. Here's an example of how to store and retrieve data using local storage:

```
    // store data in local storage
localStorage.setItem("username", "john");

// retrieve data from local storage
var username = localStorage.getItem("username");
console.log(username); // outputs "john"
```

In this example, we use the **setItem()** method to store the key-value pair "username" and "john" in local storage. We then use the **getItem()** method to retrieve the value of the "username" key and store it in a variable.

Web storage provides a convenient way to store data on the client-side and can be used to create more responsive and interactive web applications. However, it's important to be mindful of the amount of data being stored and to always consider the privacy implications of storing data on a user's device.

Understanding Geolocation

Geolocation is a feature of modern web browsers that allows web developers to retrieve the geographical location of a user's device. This can be useful for a variety of applications, such as providing location-based services, tracking the movements of a user, or customizing the content of a web application based on the user's location.

To use geolocation in HTML5, you can use the **navigator.geolocation** object in JavaScript. Here's an example of how to retrieve the user's current location:

```
// retrieve the user's location
navigator.geolocation.getCurrentPosition(function(position) {
  var latitude = position.coords.latitude;
  var longitude = position.coords.longitude;
  console.log("Latitude: " + latitude + ", Longitude: " + longitude);
});
```

In this example, we use the **getCurrentPosition()** method to retrieve the user's current position. The method takes a callback function as an argument, which is called with a **Position** object containing the latitude and longitude coordinates of the user's location.

It's important to note that geolocation is a sensitive feature that requires the user's permission to access. Most modern web browsers will prompt the user for permission before allowing a web application to access their location. As a web developer, it's important to be transparent about how you're using a user's location data and to respect their privacy.

Understanding Web Workers

Web workers are a feature of modern web browsers that allow web developers to run scripts in the background of a web page without interrupting the user interface. This can be useful for performing computationally intensive tasks or long-running operations without blocking the main thread of the web page.

Web workers work by running scripts in a separate thread from the main thread of the web page. This separate thread can be used to perform tasks such as data processing, image manipulation, or network requests. When the web worker has completed its task, it sends a message back to the main thread with the result of its computation.

To use web workers in HTML5, you can use the **Worker** object in JavaScript. Here's an example of how to create a web worker:

```
// create a new web worker
var worker = new Worker("worker.js");

// send a message to the web worker
worker.postMessage("hello");

// receive a message from the web worker
worker.onmessage = function(event) {
  console.log("Received message from worker: " + event.data);
};
```

In this example, we create a new web worker by passing the name of a JavaScript file ("worker.js") to the **Worker** constructor. We then send a message to the web worker using the **postMessage()** method and receive a message from the web worker using the **onmessage** event handler.

Web workers provide a powerful way to run scripts in the

background of a web page without interrupting the user interface. However, it's important to note that web workers have some limitations, such as not being able to access the DOM or interact with the main thread of the web page. As a web developer, it's important to consider these limitations when deciding whether or not to use web workers in your web application.

Understanding Drag and Drop

Drag and drop is a user interface feature in HTML5 that allows users to move elements on a web page by clicking and dragging them to a new location. Drag and drop is commonly used for tasks such as rearranging items in a list or moving images on a canvas.

To enable drag and drop in HTML5, you can use the **draggable** attribute on elements that you want to be draggable. Here's an example:

```
<div draggable="true">Drag me!</div>
```

In this example, we've added the **draggable** attribute to a **div** element to make it draggable. When the user clicks and drags the **div**, the browser will display a "ghost" image of the element that the user can drag around the screen.

You can also use the **ondragstart**, **ondrag**, and **ondragend** events to control the behavior of the draggable element. Here's an example:

```
<div draggable="true" ondragstart="dragStart(event)" ondrag="dragging(event)" ondragend="dragEnd(event)">Dra

<script>
function dragStart(event) {
  // do something when the user starts dragging the element
}

function dragging(event) {
  // do something while the user is dragging the element
}

function dragEnd(event) {
  // do something when the user stops dragging the element
}
</script>
```

In this example, we've added event handlers for the **ondragstart**, **ondrag**, and **ondragend** events to the **div** element. These event handlers allow us to control the behavior of the draggable element, such as by changing the appearance of the "ghost" image or updating the position of the element as the user drags it around the

screen.

Drag and drop is a powerful user interface feature that can greatly enhance the interactivity of web applications. However, it's important to ensure that drag and drop is implemented in a way that is accessible and usable for all users, including those who may have difficulty using a mouse or touch screen.

Understanding Web Sockets

Web Sockets is a protocol in HTML5 that enables two-way communication between a client and a server over a single, long-lived connection. This allows for real-time, event-driven communication between the client and server, and is ideal for applications such as online gaming, chat applications, and financial trading platforms.

Web Sockets use the WebSocket API, which is built into modern web browsers and provides a simple interface for creating and managing WebSocket connections. Here's an example of how to create a WebSocket connection in JavaScript:

```
<script>
var socket = new WebSocket("ws://example.com/socket");

socket.onopen = function(event) {
  // do something when the connection is opened
}

socket.onmessage = function(event) {
  // handle incoming messages from the server
}

socket.onclose = function(event) {
  // do something when the connection is closed
}

</script>
```

In this example, we create a new WebSocket connection to a server at **ws://example.com/socket**. We then define event handlers for the **onopen**, **onmessage**, and **onclose** events, which are triggered when the connection is opened, when incoming messages are received from the server, and when the connection is closed,

respectively.

Once the WebSocket connection is established, you can use the **send()** method to send messages to the server, and the server can use the same WebSocket connection to send messages back to the client. This allows for real-time, bidirectional communication between the client and server.

Accessibility and SEO

Accessibility and SEO are two important factors to consider when developing websites with HTML. Accessibility refers to designing and developing websites that are accessible to all users, including those with disabilities or impairments. SEO, or search engine optimization, refers to the process of optimizing a website's content and structure to improve its visibility and ranking in search engine results pages (SERPs).

Here are some tips for improving accessibility and SEO in HTML:

Accessibility:

- Use semantic HTML elements, such as headings, lists, and tables, to structure content in a meaningful and accessible way.
- Provide alternative text descriptions for images, videos, and other non-text content, using the "alt" attribute in HTML.
- Use descriptive link text that accurately reflects the destination of the link.
- Ensure that color is not the only means of conveying information, as users with visual impairments may not be able to distinguish between different colors.
- Use keyboard navigation and ensure that all interactive elements can be accessed and activated using only the keyboard.
- Provide captions or transcripts for audio and video content.

SEO:

- Use descriptive and relevant titles for pages and headings, using H1, H2, and other heading tags to

structure content.
- Use meta descriptions to provide a summary of the content on each page.
- Use descriptive URLs that accurately reflect the content of the page.
- Use keywords and phrases relevant to the content of the page in headings, content, and meta data.
- Ensure that the website is mobile-friendly and responsive, as mobile optimization is a key factor in SEO.
- Use structured data markup to provide additional information to search engines about the content of the page.

Making your HTML5 Code Accessible

Making HTML5 code accessible means designing and developing web content that can be used by people with disabilities. Accessibility in HTML5 is achieved by using best practices to optimize your code for use with assistive technologies, such as screen readers or braille displays.

Here are some tips for making your HTML5 code accessible:

- **Use semantic HTML elements**: Use headings, paragraphs, lists, and tables to structure content in a meaningful way. These elements help screen readers to interpret the content more accurately.
- **Provide text alternatives**: Use the "alt" attribute to provide text descriptions for images, videos, and other non-text content. This allows users who are visually impaired to understand the content of the page.
- **Use descriptive link text**: Use descriptive text for links, rather than generic text like "click here." This helps users understand the purpose of the link.
- **Use descriptive labels**: Use descriptive labels for form controls, such as text fields, radio buttons, and checkboxes. This allows screen readers to accurately describe the form to users.
- **Use color carefully**: Do not use color as the only means of conveying information. Ensure that all important

information is conveyed through text or other visual cues.
- **Provide keyboard navigation**: Ensure that all interactive elements can be accessed and activated using only the keyboard.
- **Test with assistive technologies**: Test your website using screen readers or other assistive technologies to ensure that it is accessible to users with disabilities.

Using Semantic Markup

Using semantic markup means using HTML elements to describe the meaning of the content on a web page, rather than using generic elements like "div" and "span" for all content. Semantic markup helps search engines and other automated systems understand the structure and meaning of the content on a web page, which can improve the page's search engine ranking and accessibility for users.

Here are some examples of semantic markup elements and when to use them:

- <**header**>: Use this element to define the header of a web page, including any navigation menus or branding.
- <**nav**>**:** Use this element to define the navigation links on a web page.
- <**main**>**:** Use this element to define the main content of a web page.
- <**article**>**:** Use this element to define a self-contained unit of content, such as a blog post or news article.
- <**section**>**:** Use this element to define a section of content that is thematically related, such as a chapter in a book or a section of a news article.
- <**aside**>**:** Use this element to define content that is related to the main content, but not central to it, such as a sidebar or advertisement.
- <**footer**>**:** Use this element to define the footer of a web page, including any copyright information or contact details.

Applying Semantic Markup

To apply semantic markup, you need to choose the appropriate

HTML element that best describes the meaning of the content you want to display on your web page. Here are some examples of how to apply semantic markup:

- Use <header> for the header section of your web page.

```
<header>
    <h1>My Website</h1>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">About</a></li>
        <li><a href="#">Contact</a></li>
      </ul>
    </nav>
  </header>
```

- Use <main> for the main content area of your web page.

```
<main>
    <article>
      <h2>Article Title</h2>
      <p>Article content goes here.</p>
    </article>
    <aside>
      <h3>Related Articles</h3>
      <ul>
        <li><a href="#">Related Article 1</a></li>
        <li><a href="#">Related Article 2</a></li>
        <li><a href="#">Related Article 3</a></li>
      </ul>
    </aside>
  </main>
```

- Use <nav> for the navigation links on your web page.

```
<nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
```

- Use <footer> for the footer section of your web page.

```
    <footer>
  <p>&copy; 2023 My Website</p>
</footer>
```

- Use <section> to group related content on your web page.

```
  <section>
      <h2>Section Title</h2>
      <p>Section content goes here.</p>
    </section>
```

Optimizing for Search Engines

Here are some tips for optimizing your HTML5 code for search engines:

- **Use descriptive page titles and meta descriptions**: Use descriptive page titles and meta descriptions to give search engines and users a clear idea of what your page is about. Include relevant keywords, but don't stuff them unnaturally.

```
<head>
    <title>My Website - Home</title>
    <meta name="description" content="A description of my website">
  </head>
```

- **Use header tags appropriately**: Use header tags (**h1**, **h2**, **h3**, etc.) to structure your content in a logical hierarchy. This helps search engines understand the

importance of different sections of your page.

- Include relevant keywords in your content: Use relevant keywords in your content, but don't overdo it. Make sure the keywords fit naturally into the content.
- **Use descriptive and relevant URLs**: Use descriptive and relevant URLs that reflect the content of your page. Avoid using long or irrelevant URLs.

```
<a href="/blog/article-name">Read the article</a>
```

- **Optimize your images**: Optimize your images by using descriptive file names and alt text, and compressing the file size for faster loading times. This can help your images show up in image searches.

```
<img src="example.jpg" alt="A description of the image">
```

- **Use internal linking:** Use internal links to connect related pages and content on your website. This helps search engines understand the organization of your website and the relationship between different pages.

```
<a href="/blog">Visit our blog</a>
```

CHAPTER SIX

HTML PRACTICAL EXERCISES

The HTML Practical Exercises chapter is a collection of practical exercises designed to help you master HTML coding skills by creating various types of websites. This chapter includes a range of projects, from simple to complex, to suit the needs of different skill levels. These projects cover a wide range of topics, including basic HTML tags and attributes, forms, tables and lists, multimedia, canvas, and SVG graphics, and advanced HTML features like web

storage, geolocation, and web workers. Each project is designed to challenge your coding abilities and help you develop your problem-solving skills. By completing these projects, you can gain a better understanding of how HTML works, improve your code quality, and create functional and visually appealing websites. Whether you are a beginner or an experienced web developer, the HTML Practical Exercises chapter can help you refine your skills and become a proficient HTML coder.

Exercise 1: Creating a Login Page

You will need to first create a folder for your code files, ensuring that both the index file and the subordinate files are together in the same folder. Do this for every single project you are carrying out.

<!DOCTYPE html>

<html>

<head>

<meta charset='utf-8'>

<meta http-equiv='X-UA-Compatible' content='IE=edge'>

<title>Example Page</title>

<meta name='viewport' content='width=device-width, initial-scale=1'>

<link rel='stylesheet' type='text/css' media='screen' href='main.css'>

<script src='main.js'></script>

</head>

<body>

<form>

<h1>Login Page</h1>

<label for="username">Username:</label>

<input type="text" id="username" name="username" required>

<label for="password">Password:</label>

<input type="password" id="password" name="password" required>

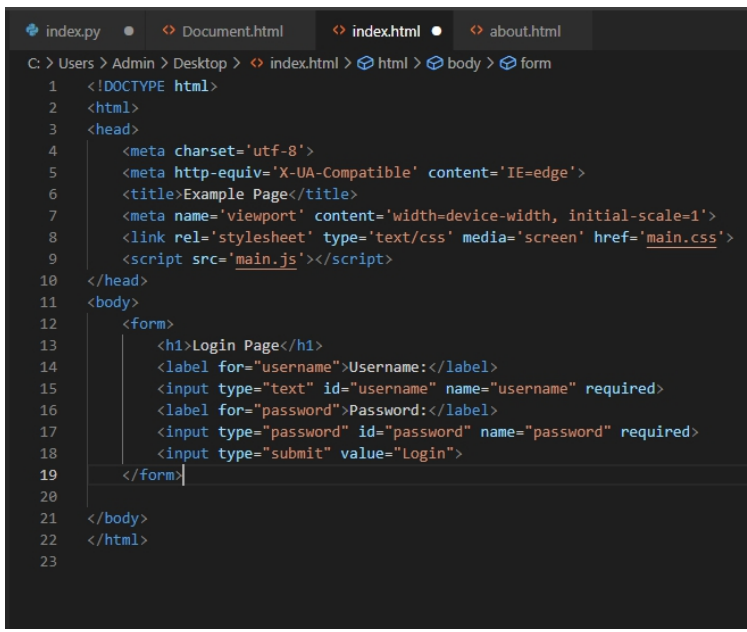
<input type="submit" value="Login">

</form>

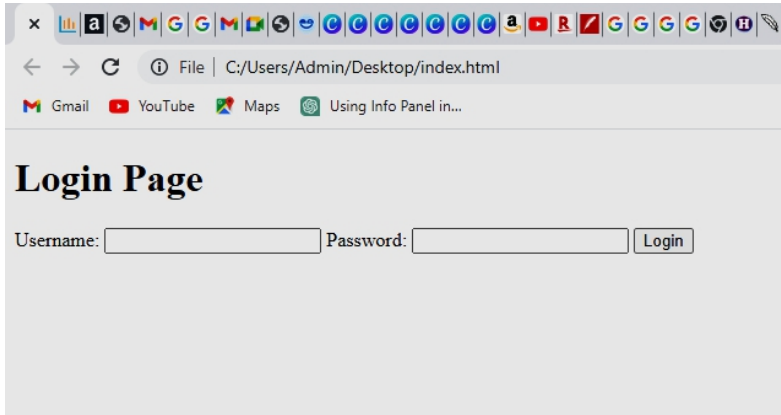
</body>

</html>

Create an HTML file with the name "**index.html**" and replicate the code above.



See the result below.



Exercise 2: Creating a Registration Page

Below is an example of a basic registration form using HTML code:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Example Page</title>
<meta name='viewport' content='width=device-width, initial-scale=1'>
<link rel='stylesheet' type='text/css' media='screen' href='main.css'>
<script src='main.js'></script>
</head>
<body>
    <h2>Registration Form</h2>
    <form>
<label for="username">Username:</label>
<input type="text" id="username" name="username" required><br><br>
```

<label for="password">Password:</label>

*<input type="password" id="password" name="password" required>

*

<label for="email">Email:</label>

*<input type="email" id="email" name="email" required>

*

<label for="dob">Date of Birth:</label>

*<input type="date" id="dob" name="dob">

*

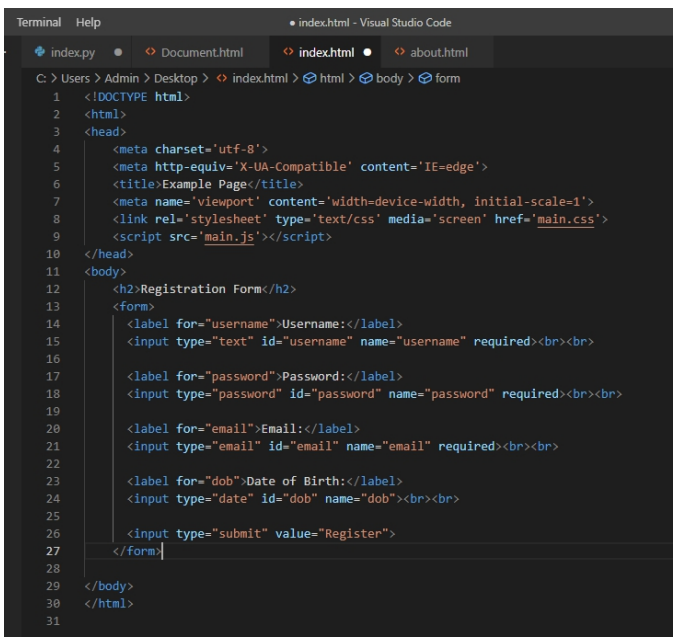
<input type="submit" value="Register">

</form>

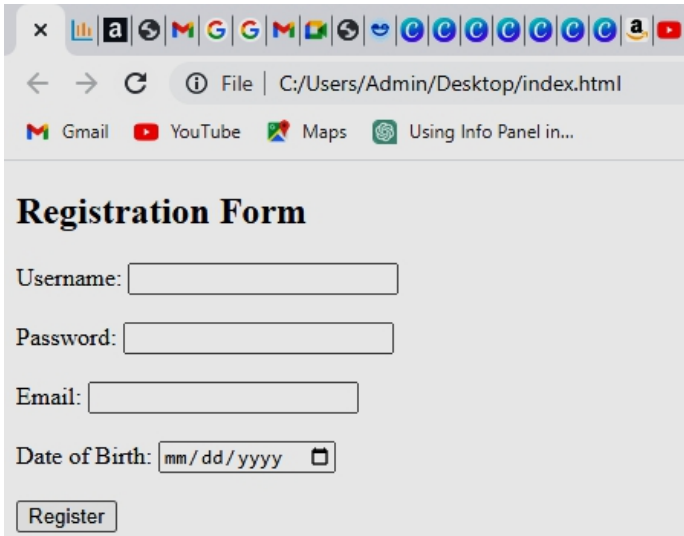
</body>

</html>

The representation of the above code in Visual Studio Code is shown below.



See the result below.



Exercise 3: Create a Simple Personal Portfolio Website

It should include the following:

- **Home page**: This page should include your name, a brief introduction about yourself, and links to your other pages.
- **About page**: This page should provide more detailed information about yourself, your skills, education, and work experience.
- **Projects page**: This page should showcase some of your past projects and include descriptions and images.
- **Contact page**: This page should include a contact form where visitors can reach out to you, as well as your social media links and any other relevant contact information.

The code below will guide you on this.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Project 3</title>
</head>
<body>
  <body>
 <header>
    <h1>Sammie Smith</h1>
    <nav>
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="projects.html">Projects</a></li>
        <li><a href="contact.html">Contact</a></li>
      </ul>
    </nav>
 </header>
 <main>
    <section>
      <h2>About Me</h2>
      <p>
```

A certified Computer Scientist, HSE professional, writer, front-end developer using (HTML, CSS, and JavaScript), can run basic programs using (Java, C++, and Visual Basic), a graphic designer, proficient in the use of Microsoft Office Suite, an accomplished teacher, and a counselor, instructing classes of different sizes and ability levels, experienced in business sales, experienced in agro extension services, can

run software installations and troubleshoot basic computer problems. An analytical thinker, self-motivated, results-driven professional with experience in leadership, organization, finance management, and people management, a fast learner that pays close attention to details and is ready to learn something new. I seek to deploy my skills and my knowledge to promote the growth and advancement of the organization and pursue her goals to realize the objectives of the organization while building a strong career for myself.

* *

</section>

<section>

<h2>My Skills</h2>

<ul>

*
Writing
*

*
Menial skill in the use of HTML,CSS, JAVA, Visual Basic and C++ Programming Languages
*

*
Proficient in Microsoft Word, Excel, Power Point Publisher, Access etc.
*

*
Fast learner, good team player, result oriented with excellent motivational skills *

*
Advance skill in the use of Corel Draw and Photoshop
*

*
*

</section>

<section>

<h2>Education</h2>

* *

* 2018-2019: The Federal Polytechnic, Bida
*

*Higher National Diploma (Upper Credit) – Computer Science.
*

*2015-2016: Niger State Polytechnic, Zungeru
*

National Diploma (Distinction) - Computer Science.

2013: Day Secondary School, Eyagi Bida

SSCE (O'Level).

2011-2013: Government Technical College, Bida

SSCE (O'Level).

2007-2010: Day Secondary School, Gaba, Niger State

JSCE

2000-2006: St. Jude's Nursery/Primary School Gaba, Niger State

First School Leaving Certificate.

</section>

<section>

<h2>Work Experience</h2>

 National Youth Service Corps: 18-05-2021 – 17-05-2022.

Taught VOC1, VOC2 & VOC3 students Computer Craft Studies (CSS), make lesson notes, organized practical classes for the students using a projector, assess the student's performances during practical sections, conducted exams for the students, run end of term school projects for the students, scanned, type and print documents.

Jiwo Academy Dutsen Kura Gwari, Minna: 10-02-2021 – 02-05-2021

 I taught the Secondary school section as an ICT subject Teacher, and organized computer practical classes for the students, I also volunteered to type the student's test questions, input the students' scores, and generate their exam report sheet using a computer program and afterward printed them.

*POS Partnership Business: 11-06-2020 – 20th-04-2021
*

*Transaction management, payment processing, sales reporting, customer relationship management, and employee management.
*

*
*

*N-power: 04-08-2018 – 30-07-2020
*

Served as Agro Extension Officer under Lavun LGA (Gaba Ward), took farmer's Data and Survey using ODK COLLECT App, sensitizing farmers on good farm practices.

*
*

*St. Jude's Nursery/Primary School, Gaba: 01-09-2013 – 05-01-2015
*

Taught the pupils of Primary Six (6) as a Class teacher taking them all subjects, taking register attendance, making lesson notes, and writing lesson plans. The school that almost closed down witnessed a great turnaround academically and financially as they yield to my advice on school management.

*
Recharge Card Retailer: 01-02-2010 – 05-01-2011
*

*
 Buying recharge cards from wholesalers, selling the recharge card to the customers, and maximizing profits.
*

*
*

*Volunteer Experience:
*

*
 Helped to upload GTC teachers' E-affidavit and other documents on their Rivers State teacher's dashboard, from 31st August to 11th September 2021.*

*
 Helped to design Divine Grace POS Berners, also helped to design jotter cover page for NCCF Opobo Zone for their batch B 2020 NCCF Send forth.*

</p>

</section>

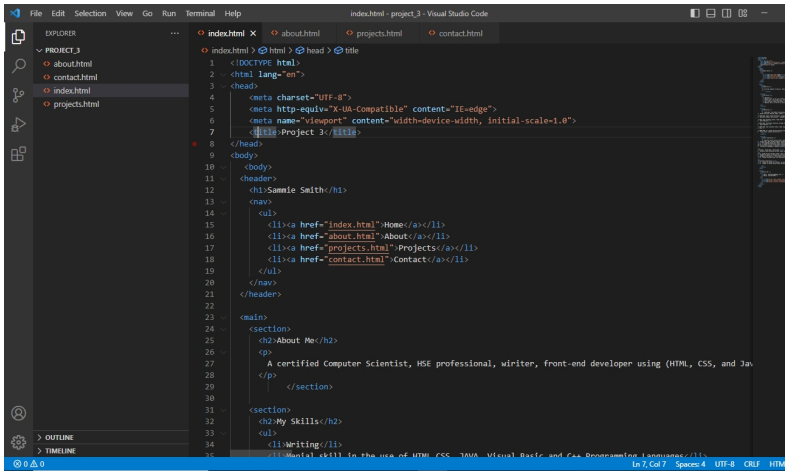
</main>

<footer>

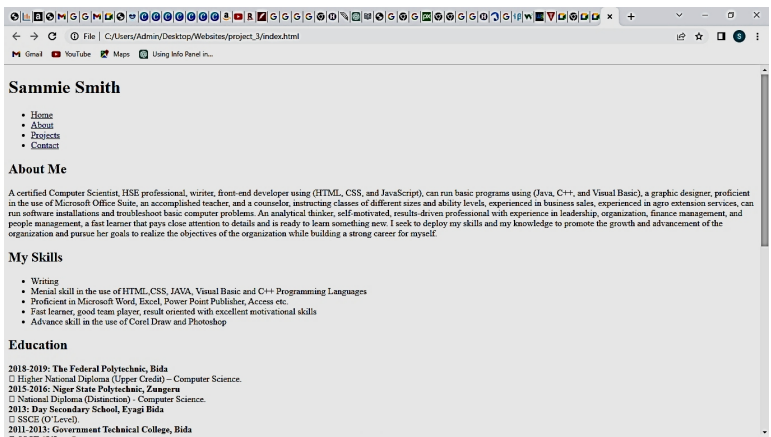
```
    <p>Contact Me:</p>
    <ul>
      <li>Email: sammiesmith@gmail.com</li>
      <li>Phone: 123-456-7890</li>
    </ul>
    <ul>
      <li><a href="https://www.linkedin.com/in/myprofile/">LinkedIn</a></li>
      <li><a href="https://github.com/myusername">GitHub</a></li>
      <li><a href="https://twitter.com/myhandle">Twitter</a></li>
    </ul>
  </footer>
</body>
</html>
```

You can create a new file for About and save it with “about.html”, Projects and save with

“projects.html”, Contact and save with “contact.html” within the same folder. Copy and paste the index file code on each of the files created, then remove the necessary elements that are not needed, and replace them with the required information. When that is done, your project would be completed. The screenshot below contains part of the code.



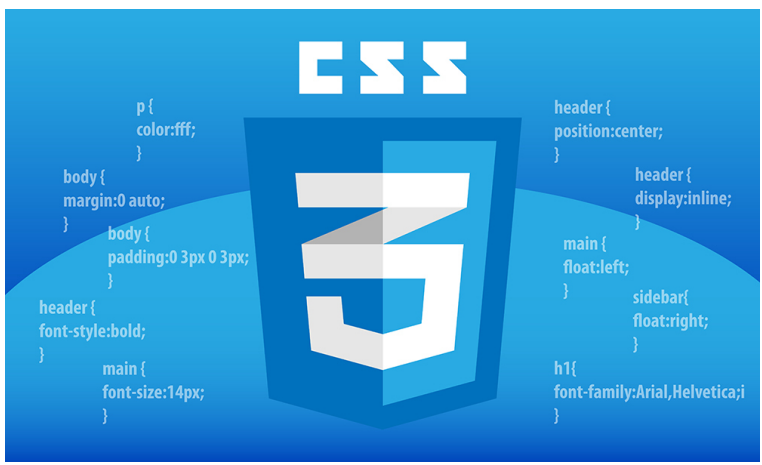
See the result from the browser below.


File | C:/Users/Admin/Desktop/Websites/project_3/index.html
Gmail YouTube Maps Using Info Panel in...
Sammie Smith
Home
About
Projects
Contact
About Me
A certified Computer Scientist, HSE professional, wiriter, front-end developer using (HTML, CSS, and JavaScript), can run basic programs using (Java, C++, and Visual Basic), a graphic designer, proficient in the use of Microsoft Office Suite, an accomplished teacher, and a counselor, instructing classes of different sizes and ability levels, experienced in business sales, experienced in agro extension services, can run software installations and troubleshoot basic computer problems. An analytical thinker, self-motivated, results-driven professional with experience in leadership, organization, finance management, and people management, a fast learner that pays close attention to details and is ready to learn something new. I seek to deploy my skills and my knowledge to promote the growth and advancement of the organization and pursue her goals to realize the objectives of the organization while building a strong career for myself.
My Skills
Writing
Menial skill in the use of HTML, CSS, JAVA, Visual Basic and C++ Programming Languages
Proficient in Microsoft Word, Excel, Power Point Publisher, Access etc.
Fast learner, good team player, result oriented with excellent motivational skills
Advance skill in the use of Corel Draw and Photoshop
Education
2018-2019: The Federal Polytechnic, Bida
Higher National Diploma (Upper Credit) – Computer Science.
2015-2016: Niger State Polytechnic, Zungeru
National Diploma (Distinction) – Computer Science.
2013: Day Secondary School, Eyagi Bida
SSCE (O'Level).
2011-2013: Government Technical College, Bida

CHAPTER SEVEN

INTRODUCTION TO CSS

CSS (Cascading Style Sheets) is a language used to describe the presentation of a document written in HTML (Hypertext Markup Language). It is used to add style and formatting to web pages and can control the layout, typography, colors, and other visual elements of a webpage. CSS can be used to create responsive designs that adapt to different screen sizes and devices, and it can be used in conjunction with HTML and JavaScript to create dynamic and interactive web pages. Understanding CSS is essential for creating professional-looking and functional websites.



What is CSS?

CSS stands for Cascading Style Sheets. It is a styling language used to describe the presentation of a document written in HTML or XML, including colors, layout, fonts, and other visual elements. CSS separates the content of a web page from its presentation, allowing for more flexible and efficient website design. By using CSS, web developers can control the look and feel of a website across multiple web pages and devices, making it easier to maintain and update.

How is CSS used in Web Development?

CSS works by selecting HTML elements and defining how they

should be displayed. For example, you can select all paragraphs on a page and set their font size, color, and line height to create a consistent look throughout the document. CSS can be applied to HTML elements directly or through classes and IDs, which allows for more targeted styling.

CSS is an essential part of modern web development and is used to create visually appealing and user-friendly websites. It allows developers to separate the content and structure of a webpage from its presentation, making it easier to maintain and update. With CSS, you can create responsive designs that adapt to different screen sizes and devices, and add animations and interactive elements to enhance the user experience.

How CSS works with HTML

CSS works with HTML by targeting and styling the HTML elements within a webpage. HTML provides the structure and content of the webpage, while CSS is used to control the presentation and layout of that content.

CSS selectors are used to target HTML elements, and then properties are applied to those elements to change their appearance. For example, the following CSS code would target all <p> elements and set their font size to 16 pixels:

```
p {
  font-size: 16px;
}
```

This would make all paragraphs within the HTML document have a font size of 16 pixels. CSS can be applied to HTML documents in a few different ways, such as through an external stylesheet file or inline styles within the HTML document.

Understand the Difference Between CSS and HTML

HTML and CSS are two separate languages used in web development. HTML, or HyperText Markup Language, is used for creating the structure and content of a web page. It defines the various elements that make up the page, such as headings,

paragraphs, images, and links.

CSS, or Cascading Style Sheets, is used for styling the presentation of the web page. It defines how the HTML elements should be displayed, such as the color, font, size, and layout. CSS can be used to create complex page layouts and visually appealing designs.

In short, while HTML defines the content and structure of a web page, CSS defines its presentation and style. They work together to create a complete and functional website.

Benefits of using CSS

There are several benefits of using CSS in web development:

- **Separation of presentation and content**: CSS allows for a clear separation between the presentation and content of a web page, making it easier to maintain and update the design without affecting the content.
- **Consistency**: CSS enables the designer to apply consistent styles to multiple pages, ensuring that the look and feel of the website are uniform.
- **Flexibility**: CSS provides flexibility in design by allowing for the use of various layout and positioning techniques, as well as the ability to create responsive designs that adapt to different screen sizes.
- **Faster page load times**: By separating the presentation and content, CSS reduces the amount of code needed to load a web page, resulting in faster page load times.
- **Accessibility**: CSS makes it easier to create accessible websites by providing the ability to apply styles to specific elements, such as headings and links, and improving the readability of the content.

Basic CSS Syntax

The basic syntax of CSS consists of a selector and a set of declarations:

```
selector {
property1: value1;
property2: value2;
}
```

The selector selects the HTML element(s) that you want to apply the styles, and the declarations define the styles themselves. Each declaration consists of a property and its value, separated by a colon.

Here is an example:

```
h1 {
color: blue;
font-size: 24px;
text-align: center;
}
```

In this example, **h1** is the selector that targets all **h1** HTML elements, and the declarations define the styles that will be applied to them: blue text color, 24px font size, and centered text alignment.

Writing CSS rules

Writing CSS rules involves selecting HTML elements and applying style properties to them. The basic syntax for writing a CSS rule is:

```
selector {
property: value;
}
```

- **Selector**: Specifies which HTML element(s) the rule applies to
- **Property**: Defines the aspect of the element that we want to style
- **Value**: Sets the value of the property

For example, to set the font color of all paragraph elements to red,

we would use the following CSS rule:

Multiple properties can be applied to the same selector, by separating them with a semicolon (;). For example:

This rule sets the font size of all h1 elements to 24 pixels, sets the color to blue, and centers the text.

Selectors and Declarations

In CSS, selectors and declarations are used to target specific HTML elements and apply styling rules to them.

Selectors are patterns used to select the elements that you want to style, while declarations define the styling rules that you want to apply to those elements. For example, consider the following CSS code:

In this example, the **h1** selector targets all <**h1**> elements on the page, and the declarations inside the curly braces apply the styling rules. The **color** declaration sets the text color to red, and the **font-size** declaration sets the font size to 24 pixels.

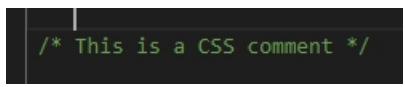
Selectors can target specific elements based on their tag name, class, ID, attribute, or a combination of these factors. There are also pseudo-classes and pseudo-elements that allow you to target elements based on their state or position within the document.

Declarations can include a variety of properties, such as color, font, background, border, margin, padding, and more. Each property can have a value assigned to it, such as a color name, a numeric value, a percentage, or a keyword.

Comments in CSS

In CSS, you can add comments to your code to make it more readable and easier to understand. Comments are not displayed on the web page and do not affect the styling of your elements.

To add comments in CSS, you can use the /* */ syntax. Anything written between /* and */ will be considered a comment and will not be executed by the browser. For example:



You can also use comments to temporarily disable a block of code without deleting it. This can be useful when testing different styles for your elements. Simply add /* at the beginning of the code you want to disable and */ at the end to re-enable it. For example:

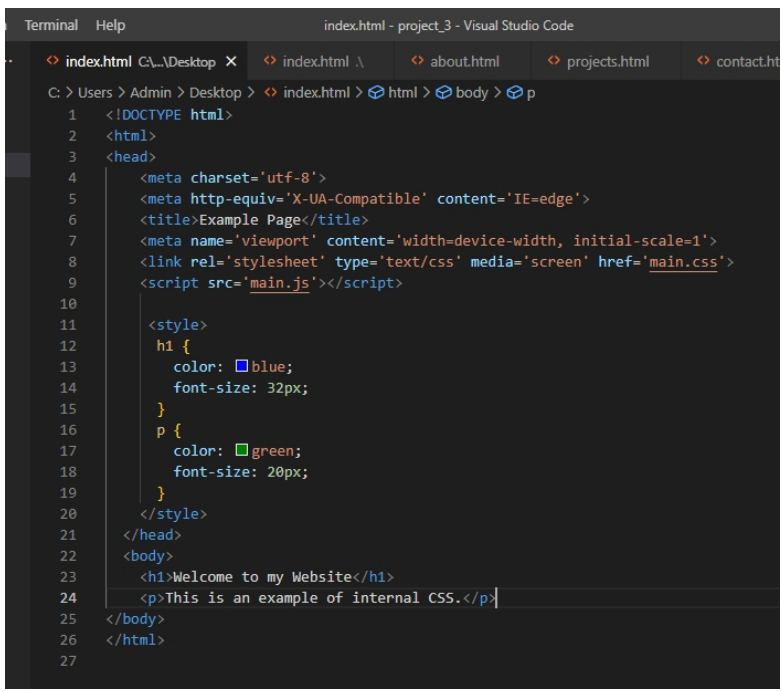
In the example above, the first block of code is commented out and will not be executed by the browser, while the second block will apply the color blue to all **h1** elements on the page.

External and Internal CSS

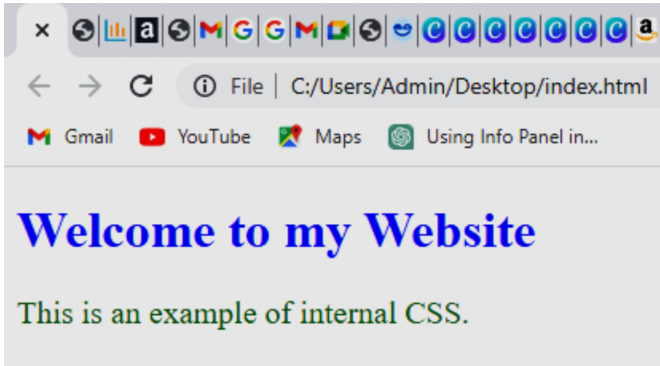
External and Internal CSS are two ways of organizing CSS code on a web page.

Internal CSS is CSS code that is placed within the head section of an HTML document, using the <style> tag. It applies only to the current web page and cannot be used on other pages on the website. Internal CSS is useful for small, single-page websites or for making quick changes to a specific page without affecting the rest of the site.

Here is an example of internal CSS code:

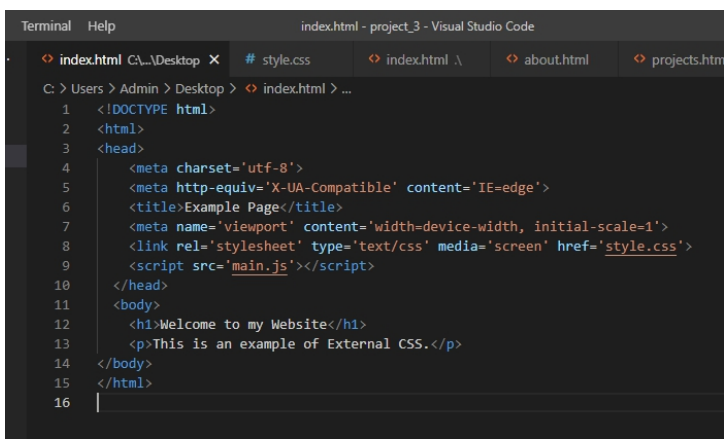


See the corresponding result below.



External CSS is CSS code that is stored in a separate .css file and linked to the HTML document using the <link> tag in the head section of the HTML document. This allows the CSS to be used across multiple web pages on a website. External CSS is useful for larger websites with multiple pages and for maintaining consistency across the site.

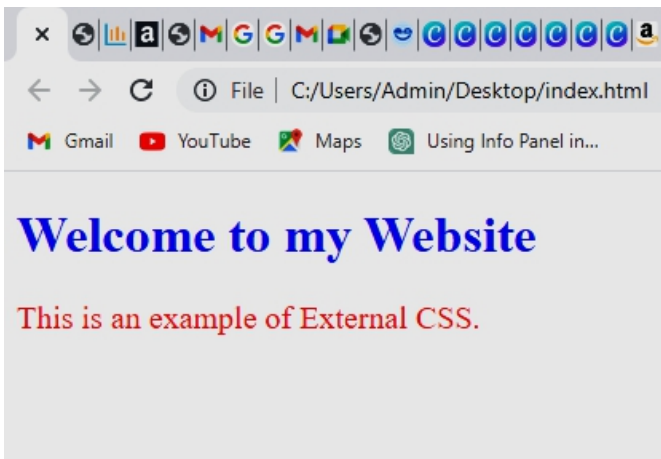
Here is an example of external CSS code:



Below is the corresponding style.css file:

```
Terminal  Help                                    style.css
index.html C:\...\Desktop        # style.css
C: > Users > Admin > Desktop > # style.css > ...
1   h1 {
2       color: blue;
3       font-size: 32px;
4   }
5   p {
6       color: red;
7       font-size: 20px;
8   }
9
```

The corresponding result is shown below.



Linking an External CSS

Linking an external CSS file to an HTML document involves the following steps:

- **Create a CSS file:** Create a new file and save it with a ".css" extension, e.g. "styles.css". This file will contain all of your CSS code.
- **Add CSS code**: Add your CSS code to the "styles.css" file. This can include any selectors, declarations, and rules that you want to apply to your HTML document.
- **Link the CSS file to your HTML document**: In the

head section of your HTML document, add a link element that specifies the location of the CSS file. The link element should look like this:

- You can replace the "styles.css" name or path with the actual path to your CSS file. This path can be a relative or absolute URL.

- Save your files: Save both your HTML and CSS files.
- After these steps, your HTML document will be linked to the external CSS file, and any styles you defined in the CSS file will be applied to your HTML elements.

CHAPTER EIGHT

CSS STYLE PROPERTIES

CSS Style Properties are essential tools for web developers to customize the appearance of HTML elements. These properties allow developers to control the size, color, font, layout, and other visual characteristics of web pages. In this chapter, we will explore the various CSS Style Properties available and how they can be used to create visually appealing websites, alongside several other CSS components. We will also look at how to combine multiple style properties to create complex designs and layouts. By the end of this chapter, you will have a good understanding of how to use CSS Style Properties to customize the look and feel of your web pages.

Text Properties

CSS Text Properties refer to the different properties that are used to style text content in HTML documents. These properties include font styles, font sizes, font weights, line heights, text alignments, and text decorations. With CSS Text Properties, you can control the appearance of text on your website, making it more visually appealing and easy to read.

In this chapter, we will explore the different CSS Text Properties and how they can be used to enhance the appearance of your website's text content. We will also provide examples and best practices for using these properties effectively, so you can create beautiful and readable text on your website. Whether you are a beginner or an experienced web developer, understanding CSS Text Properties is essential for creating great web content.

What are CSS Text Properties?

Text properties in CSS are used to style and format the content of text elements within an HTML document. These properties allow developers to control the font family, font size, font weight, line height, text alignment, text decoration, and text transformation.

By using text properties in CSS, you can enhance the readability and appearance of your web pages. For example, you can change the font size to make the text more legible, adjust the line height to improve the spacing between lines, and add text decoration to highlight important text.

Some of the commonly used text properties in CSS include font-family, font-size, font-weight, line-height, text-align, text-decoration, and text-transform. In the following sections, we will explore these properties in more detail and learn how to use them to create effective and attractive web designs.

Applying Text Properties

To apply text properties in CSS, you can use the following syntax:

```
selector {
  property1: value1;
  property2: value2;
  ...
}
```

Here's an example of applying text properties to an **h1** element:

```
h1 {
  font-family: Arial, sans-serif;
  font-size: 32px;
  font-weight: bold;
  color: #333;
  text-align: center;
  text-transform: uppercase;
  line-height: 1.2;
  letter-spacing: 1px;
}
```

In this example, we set the font family to Arial or any sans-serif font, font size to 32 pixels, font-weight to bold, color to #333, text alignment to center, text transformation to uppercase, line height to 1.2 times the font size, and letter spacing to 1 pixel.

These text properties can be applied to any HTML element, such as paragraphs, headings, and lists, to change their appearance and improve the readability of the text.

Font Properties

Font properties in CSS are used to control the size, style, weight, and family of the text. Here are some common font properties:

- **font-size**: sets the size of the font. It can be specified in pixels, ems, rems, or other units.
- **font-style**: sets the style of the font, such as italic, oblique, or normal.
- **font-weight**: sets the weight of the font, such as bold, bolder, lighter, or normal.

- **font-family**: sets the font family for the text, such as Arial, Times New Roman, or Verdana.
- **text-transform**: transforms the text to uppercase, lowercase, or capitalized.
- **text-decoration**: adds decoration to the text, such as underline, overline, line-through, or none.

Applying Font Properties

To apply these font properties, you can use CSS rules with selectors to target specific HTML elements, and then add the desired font properties to the declarations within the rules. For example:

```
h1 {
  font-size: 36px;
  font-weight: bold;
  font-family: Arial, sans-serif;
  text-transform: uppercase;
  text-decoration: underline;
}
```

This rule applies to all <**h1**> elements and sets the font size to 36 pixels, the font weight to bold, the font family to Arial or sans-serif (if Arial is not available), the text-transform to uppercase, and the text-decoration to underline.

Color Properties

Color properties in CSS allow you to specify the color of text, background, borders, and other elements on a webpage. There are several ways to define colors in CSS, including:

- **Named colors:** CSS has predefined names for 140 different colors, such as "red", "blue", "green", etc.
- **Hexadecimal notation**: This method allows you to specify a color using a six-digit code that represents the amount of red, green, and blue in the color.
- **RGB notation**: This method specifies the amount of red, green, and blue in a color using three numbers between 0 and 255.
- **HSL notation**: This method specifies the color using its hue, saturation, and lightness values.

In addition to specifying colors for specific elements, you can also use CSS to apply color to specific parts of an element, such as its border or background. CSS also allows you to apply transparency to colors using the "opacity" property.

Applying Color properties

To apply color properties in CSS, you can use different color models like RGB, RGBA, Hexadecimal, HSL, HSLA, etc. Here are some examples:

- Using RGB values:

```
color: rgb(255, 0, 0); /* red */
background-color: rgb(255, 255, 0); /* yellow */
```

- Using Hexadecimal values:

```
color: #FF0000; /* red */
background-color: #FFFF00; /* yellow */
```

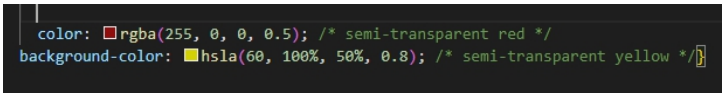
- Using HSL values:

```
color: hsl(0, 100%, 50%); /* red */
background-color: hsl(60, 100%, 50%); /* yellow */
```

- You can also apply color to specific elements using selectors:

```
h1 {
  color: #FF0000;
}

p {
  color: rgb(0, 128, 0);
}
```

- You can also set the opacity of colors using RGBA and HSLA values:



In addition to text color and background color, you can also use color properties to set border color, box shadow color, and other visual effects in CSS.

Background Properties

related attributes of an element. These properties allow designers to customize the background of a webpage or specific sections of a webpage to make it more visually appealing.

The background properties in CSS include:

- **background-color**: This property is used to set the background color of an element.
- **background-image**: This property is used to set the background image of an element.
- **background-repeat**: This property is used to specify whether or not the background image should be repeated vertically, horizontally, or not at all.
- **background-position**: This property is used to set the starting position of the background image.
- **background-size**: This property is used to set the size of the background image.
- **background-attachment**: This property is used to specify whether the background image should be fixed or scroll with the content of the element.

CSS Box Model

The CSS box model is a layout concept that defines the rectangular boxes that are generated for HTML elements in a web page. Each box consists of four parts: content, padding, border, and margin. Understanding the box model is essential for creating well-designed and visually appealing web pages.

The content area is the space where the actual content of an element, such as text or an image, is displayed. The padding is the space between the content and the border. The border is the line that surrounds the content and padding. The margin is the space

between the border and the next element on the page.

The box model is important in CSS because it affects the layout and spacing of elements on a page. By adjusting the padding, border, and margin properties of an element, web developers can control the space around and between elements. This can help create a consistent and visually pleasing layout for a web page.

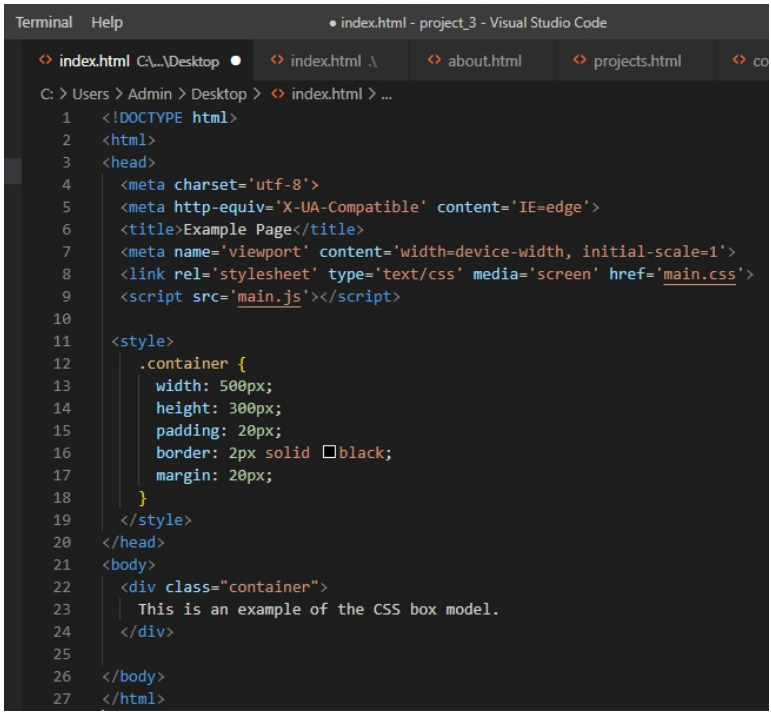
CSS provides several properties for adjusting the box model, including padding, border, and margin. These properties can be set individually for each side of the box, or all sides at once. Additionally, the box-sizing property can be used to control how the width and height of an element are calculated, including whether or not they include padding and border.

Applying CSS Box Model

To apply the CSS box model, you can use the following properties:

- **Width and height**: You can set the width and height of an element using the **width** and **height** properties. By default, the width and height of an element are set to **auto**.
- **Padding**: You can add padding to an element using the **padding** property. Padding is the space between the element's content and its border.
- **Border**: You can add a border to an element using the **border** property. The border surrounds the element's padding and content.
- **Margin**: You can add a margin to an element using the **margin** property. Margin is the space between the element's border and the adjacent elements.

Here's an example of how to apply the CSS box model:



In this example, we have a **div** element with the class **container**. We have set its width to **500px**, height to **300px**, added **20px** of padding, a **2px** solid black border, and **20px** of margin. The content of the **div** element is "This is an example of the CSS box model."

Understanding the Box Model

The CSS box model is a fundamental concept in web design that explains how elements are structured on a web page. Every HTML element is considered a box, and the box model defines how those boxes are laid out, sized, and styled.

The box model consists of four parts: content, padding, border, and margin. The content is the actual text, image, or other media that the element contains. The padding is the space between the content and the border. The border is a line or area that surrounds the padding and content. The margin is the space between the border and the neighboring elements.

The size of each box is determined by the content plus the padding, border, and margin. For example, if an element has a width of 100

pixels and a padding of 10 pixels, the total width of the box would be 120 pixels (100 pixels for the content and 10 pixels of padding on either side).

Margin, Border, and Padding

Margin, border, and padding are three important properties of the CSS box model. They determine the size and spacing of an element on a web page.

- **Margin**: The space outside the border of an element. It creates a gap between two adjacent elements. You can use a margin to control the amount of white space around an element.
- **Border**: The line that surrounds an element. It can be set to a specific width, style, and color. A border can be used to visually separate an element from other content.
- **Padding**: The space between the content of an element and its border. You can use padding to add space between the content and the border or to create a visual effect within an element.

All of these properties can be set using CSS. You can specify values for each of them separately or use the shorthand property to set them all at once. For example, the following code sets a margin of 10 pixels on all sides, a border with a width of 1 pixel, and a padding of 5 pixels inside the border:

```
.my-element {
  margin: 10px;
  border: 1px solid black;
  padding: 5px;
}
```

Width and Height

In CSS, the width and height properties are used to set the dimensions of an element. The **width** property is used to set the width of an element, and it can take a value in pixels, percentages, ems, or other CSS length units. For example:

```
div {
  width: 200px;
}
```

The above CSS rule sets the width of all **div** elements to 200 pixels. The **height** property is used to set the height of an element, and it can also take a value in pixels, percentages, ems, or other CSS length units. For example:

```
div {
height: 100px;
}
```

The above CSS rule sets the height of all **div** elements to 100 pixels. When the **width** and **height** properties are used together with the box model properties (**padding**, **border**, and **margin**), they determine the total size of an element on the web page.

CSS Layouts

CSS layouts refer to the arrangement of elements on a web page. A good layout is essential for creating an attractive and easy-to-use website. CSS provides a wide variety of tools for creating layouts, including:

- **Display property**: This property determines the type of box an element generates. The most commonly used values are block, inline, and inline-block.
- **Positioning**: This property determines the position of an element on a page. There are four values: static, relative, absolute, and fixed.
- **Float**: This property moves an element to the left or right of its container.
- **Clear**: This property is used to clear floated elements.
- **Flexbox**: This is a new layout module introduced in CSS3. It is a powerful tool for creating flexible and responsive layouts.

- **Grid**: This is another layout module introduced in CSS3. It allows developers to create complex grid-based layouts.

CSS Floats

Float is a CSS property that allows an element to be positioned to the left or right of its parent container, allowing other content to flow around it. This property is commonly used for creating column layouts and image galleries.

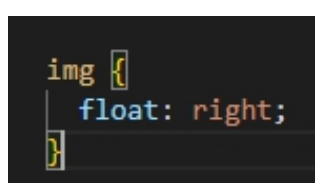
When an element is floated, it is taken out of the normal document flow, meaning that other elements will flow around it. To use the float property, you simply apply it to the element that you want to float, and then set the width and margin properties to position it correctly.

Floats can be useful for creating complex layouts, but they can also cause problems if not used correctly. For example, floated elements may overlap or push other content out of position, and they can also be difficult to work with when it comes to responsive design.

To avoid these issues, it's important to use clearfix techniques to clear floats and ensure that your layout remains stable across different devices and screen sizes. Additionally, CSS Grid and Flexbox layouts provide more modern and flexible alternatives to using floats for layout design.

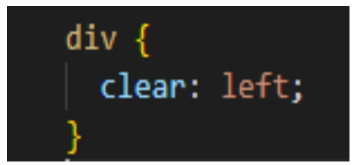
Applying Float

To apply a float in CSS, you need to use the **float** property. The value of the property can be **left** or **right** to specify which direction the element should be floated. For example, if you want to float an image to the right of a paragraph of text, you would use the following CSS:



This will cause the image to float to the right, allowing text to flow around it on the left.

It's important to note that when you float an element, it will be taken out of the normal flow of the document, which can have an impact on the layout of other elements. To prevent this, you can use the **clear** property to specify that no floating elements should appear on a particular side of the element. For example, to clear all floats to the left of an element, you would use the following CSS:



This will cause the element to be placed below any elements that have been floated to the left.

Positioning

Positioning in CSS refers to how an element is placed on a web page, either with the browser window, the containing element, or other elements on the page. There are four different types of positioning in CSS:

- **Static Positioning**: This is the default positioning for all HTML elements, where the element appears in the order it is written in the HTML code.
- **Relative Positioning**: This type of positioning allows an element to be moved from its normal position by a specified number of pixels or percentage values, while still taking up its original space in the document flow.
- **Absolute Positioning**: This type of positioning allows an element to be positioned relative to its closest positioned ancestor element, or if none exists, relative to the initial containing block. It is removed from the normal document flow and will not affect the position of other elements on the page.
- **Fixed Positioning**: This type of positioning is similar to absolute positioning, but the element is positioned relative to the browser window rather than its containing element. The element remains in the same position even if the user scrolls the page.

By using CSS positioning, web developers can create complex

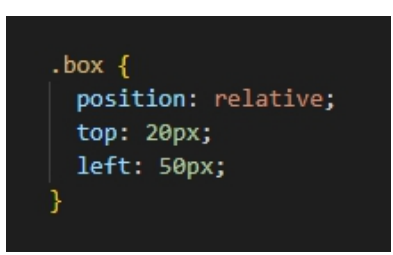
layouts and positioning effects on their web pages. However, it is important to use positioning judiciously, as it can be difficult to maintain consistency across different screen sizes and devices.

Applying Positioning

To apply positioning in CSS, you can use the **position** property. There are four different values you can use for the **position** property:

- **static**: This is the default value, and it means that the element will flow in the normal document flow.
- **relative**: This value positions the element relative to its normal position in the document flow. You can then use the **top**, **bottom**, **left**, and **right** properties to move the element to its normal position.
- **absolute**: This value positions the element relative to its closest positioned ancestor. If there is no positioned ancestor, the element will be positioned relative to the document body. You can then use the **top**, **bottom**, **left**, and **right** properties to move the element with its closest positioned ancestor.
- **fixed**: This value positions the element relative to the viewport, meaning that it will always stay in the same position even if the page is scrolled. You can then use the **top**, **bottom**, **left**, and **right** properties to move the element with the viewport.

Below is an example of how to use the **position** property:



This will position the element with the class **box** 20 pixels down and 50 pixels to the right of its normal position in the document flow.

Display

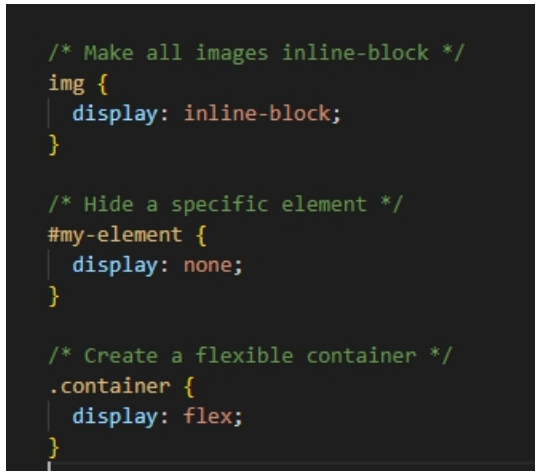
In CSS, the **display** property is used to define the type of box used to represent an HTML element. It has a variety of values that control how the element is displayed and interacts with other elements on the page.

The most common **display** values are:

- **block**: The element is displayed as a block-level element, with a line break before and after the element. This is the default display value for elements such as <**div**> and <**p**>.
- **inline**: The element is displayed as an inline-level element, with no line break before or after the element. This is the default display value for elements such as <**span**> and <**a**>.
- **inline-block**: The element is displayed as an inline-level block container, with a line break before and after the element. This is useful for elements that need to be both inline and have a width or height, such as images or form controls.
- **none**: The element is not displayed on the page at all. This is often used for elements that are hidden based on user interaction or for accessibility purposes.
- **flex**: The element is displayed as a flex container, which allows for easy creation of flexible layouts. This is often used for responsive web design.
- **grid**: The element is displayed as a grid container, which allows for more complex layouts than the **flex** property. This is also used for responsive web design.
- **table**: The element is displayed as a table element, with the same behavior as the <**table**> HTML element. This is useful for creating layouts with a fixed number of columns or rows.

Applying Display

By using the **display** property, you can control the layout of your web pages and create more effective designs. Below is an example of how to apply the **display** property:



```css
/* Make all images inline-block */
img {
  display: inline-block;
}

/* Hide a specific element */
#my-element {
  display: none;
}

/* Create a flexible container */
.container {
  display: flex;
}
```

Understanding CSS Grid

CSS Grid is a two-dimensional layout system that allows developers to create complex grid layouts with ease. It is a relatively new addition to CSS, first introduced in 2017, and has quickly become a popular way to create responsive and flexible layouts.

With CSS Grid, developers can define rows and columns and then place content within those areas. This allows for a high degree of flexibility in layout design, as elements can be placed anywhere within the grid, regardless of their order in the HTML document.

CSS Grid also supports responsive design, allowing for easy reordering of content and the ability to change the layout based on screen size or device orientation.

Some key features of CSS Grid include:

- Two-dimensional grid layouts
- Ability to define rows and columns
- Placing content within grid areas
- Support for responsive design
- Control over the sizing and spacing of grid elements

Applying CSS Grid

To apply CSS Grid, you need to follow these steps:

- Define the parent container as a grid container by

setting the display property to the grid.

- Specify the number of rows and columns for the grid using the grid-template-rows and grid-template-columns properties.
- Define the size and spacing of the grid items using the grid-template-rows and grid-template-columns properties.
- Position the grid items within the grid using the grid-row and grid-column properties.

Below is an example of how to apply CSS Grid:

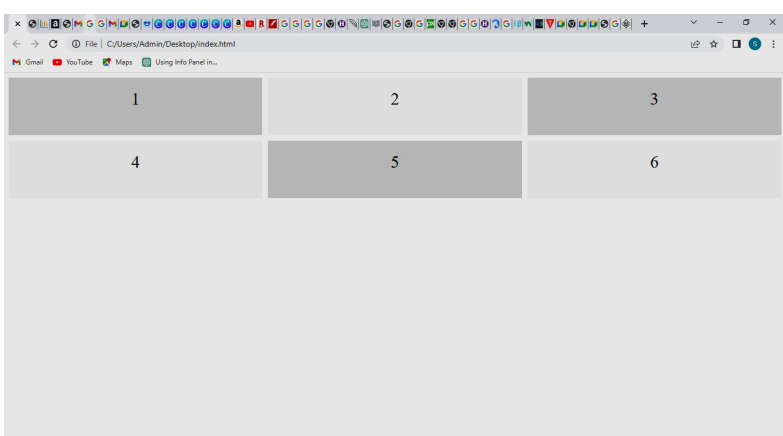
HTML code:

```
    <div class="grid-container">
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
</div>
```

CSS code:

```css
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(2, 100px);
  grid-gap: 10px;
}

.grid-item {
  background-color: #f2f2f2;
  padding: 20px;
  font-size: 30px;
  text-align: center;
}

.grid-item:nth-child(odd) {
  background-color: #ccc;
}
```

In this example, we have created a grid container with 2 rows and 3 columns using the **grid-template-rows** and **grid-template-columns** properties. We have also set the size and spacing of the grid items using the **grid-template-rows** and **grid-template-columns** properties and positioned them within the grid using the **grid-row** and **grid-column** properties. The result will be a 2x3 grid with 6 grid items inside. The result of the HTML and CSS code above is shown below.



CSS Animations

CSS Animations refer to the process of adding motion or movement to elements on a web page using CSS (Cascading Style Sheets). This is done by defining specific animations using CSS rules, which can be applied to different HTML elements such as text, images, buttons, and other page elements.

Animations can be used to create engaging and interactive effects on a web page, such as transitions, fades, rotations, and more. They can also be used to add visual interest and enhance the user experience. CSS Animations are typically defined using the **@keyframes** rule, which allows you to specify different stages of animation and the CSS properties that should be applied at each stage.

Some of the benefits of using CSS Animations include:

- They are lightweight and don't require additional JavaScript or other libraries
- They can be easily customized and adjusted to fit your specific needs
- They are supported by all modern web browsers
- They can help improve the overall user experience and engagement on a website.

When using CSS Animations, it's important to consider factors such as performance, browser compatibility, and accessibility. Animations should be used thoughtfully and sparingly, and should not interfere with the functionality of the website or make it difficult for users to navigate or interact with the content.

Applying CSS Animations

To apply CSS Animations, you need to follow these steps:

- Define the keyframes: Keyframes are used to define the animation. You can define the animation at different stages using percentage values.
- Assign the keyframes to an element: You can assign the keyframes to an element using the animation-name property.

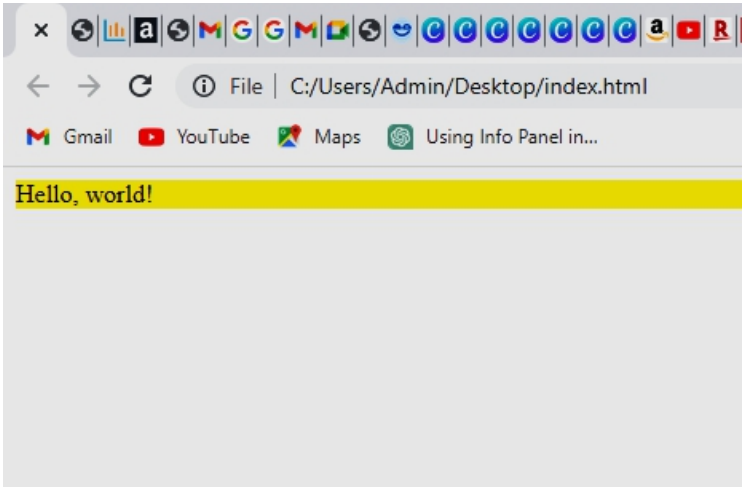
- Set the duration and timing function: You can set the duration of the animation and timing function using the animation-duration and animation-timing-function properties.
- Specify the animation direction and iteration count: You can specify the direction of the animation and the number of times the animation should be repeated using the animation-direction and animation-iteration-count properties.

Here is an example of how to apply a simple CSS animation:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Example Page</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' type='text/css' media='screen' href='main.css'>
    <script src='main.js'></script>
 <style>
     /* Define the keyframes */
     @keyframes example {
       0%   {background-color: red;}
       50%  {background-color: yellow;}
       100% {background-color: blue;}
     }

     /* Assign the keyframes to an element */
     div {
       animation-name: example;
       animation-duration: 4s;
       animation-timing-function: ease;
       animation-delay: 2s;
       animation-iteration-count: infinite;
       animation-direction: alternate;
     }
   </style>
  </head>
  <body>
    <div>Hello, world!</div>


</body>
</html>
```

In this example, we define a keyframe called **example** that changes the background color of the element from red to yellow to blue throughout the animation. We then assign this keyframe to a **div** element and specify the duration, timing function, delay, iteration count, and direction of the animation using the relevant CSS

properties. See the web result below.



CSS transitions

CSS transitions are a way to create smooth and gradual animations between two states of an element. They allow you to define how an element should change its properties over a specified duration and timing function.

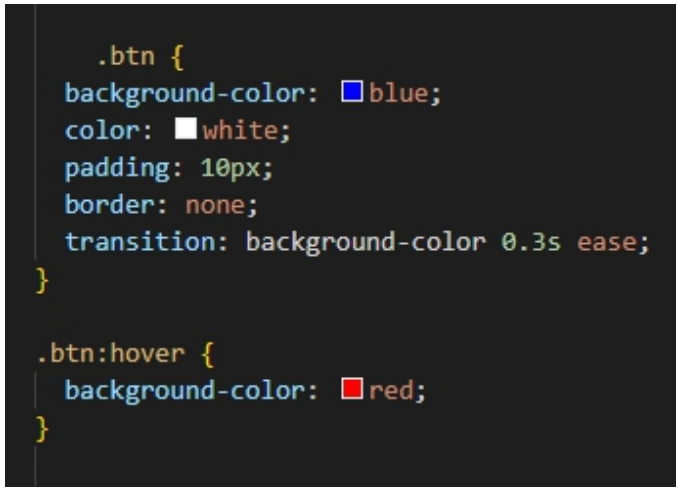
To apply a CSS transition, you need to first define the starting state of the element using CSS properties. Then, you can use the **transition** property to specify the duration, timing function, and other parameters of the animation.

Here's an example of a CSS transition that changes the background color of a button when it's hovered over:

HTML Code:

```
<button class="btn">Hover me!</button>
```

CSS Code:



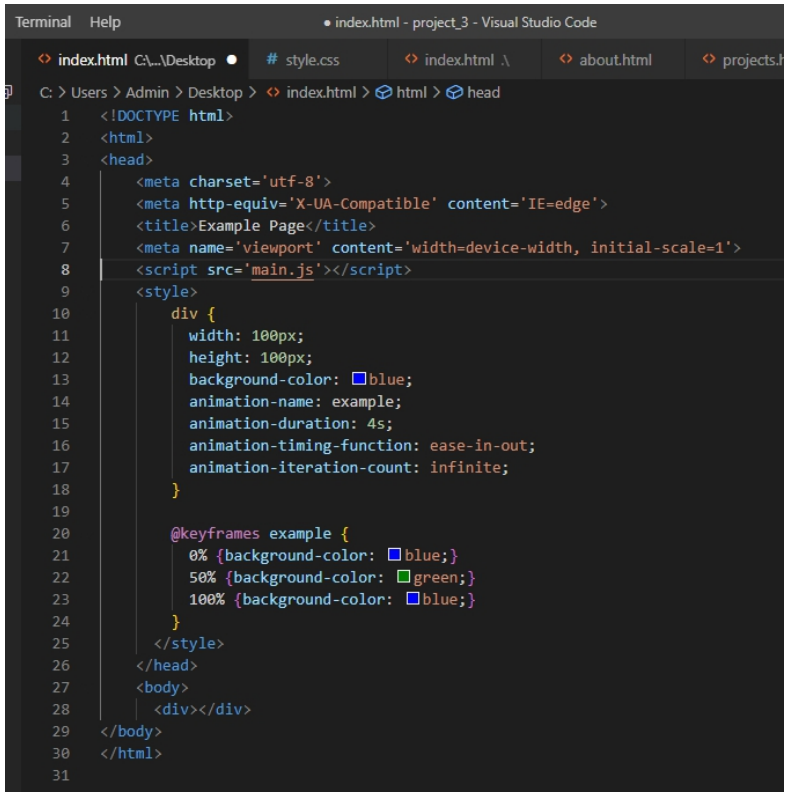
In this example, we define a button with a blue background color and white text. We also set the **transition** property to **background-color 0.3s ease**, which means that the transition will last for 0.3 seconds and will use the **ease** timing function.

When the button is hovered over, we apply a new background color of red using the **:hover** pseudo-class. Because of the **transition** property, the background color change will be smooth and gradual over the specified duration and timing function. You can apply CSS transitions to a wide range of CSS properties, including background color, font size, width, height, opacity, and more.

Animating with Keyframes

Animating with keyframes is a CSS technique that allows you to create complex animations with more control than traditional transitions. It involves defining the animation steps at different points in time, using the **@keyframes** rule, and then applying that animation to an element using the **animation** property.

Here's an example of using keyframes to create a simple animation:



In this example, we've defined a **div** element with a blue background color and a set width and height. We then apply the **example** animation to it using the **animation-name** property. We specify the duration of the animation, the easing function, and how many times it should repeat using the **animation-duration**, **animation-timing-function**, and **animation-iteration-count** properties, respectively.

The actual animation is defined in the **@keyframes** rule. Here, we define the animation steps at different points in time: at 0%, the background color is blue, at 50% it's green, and at 100% it's blue again. By using the **animation** property to apply the **example** animation to the **div**, we get a looping animation where the background color fades between blue and green.

Undesratanding Flexbox

Flexbox is a layout module in CSS that provides a flexible way to

layout, align and distribute space among items in a container. With Flexbox, you can create flexible and responsive layouts that can adapt to different screen sizes and orientations.

Flexbox consists of two important components: the container (also known as the flex container) and the items (also known as the flex items). The container is used to group and control the layout of the items inside it. The items are the individual elements inside the container that can be arranged and aligned using flexbox properties.

Some of the key flexbox properties include:

- **display: flex** - specifies that the container should use flexbox layout
- **flex-direction** - specifies the direction of the main axis (row or column) along which the items should be laid out
- **justify-content** - specifies how the items should be aligned along the main axis (e.g. center, start, end, space-between, etc.)
- **align-items** - specifies how the items should be aligned along the cross axis (e.g. center, start, end, stretch, etc.)
- **flex-wrap** - specifies whether the items should wrap to the next line if there is not enough space on a single line
- **flex-grow** - specifies how much an item should grow relative to other items in the container
- **flex-shrink** - specifies how much an item should shrink relative to other items in the container
- **flex-basis** - specifies the initial size of an item before any remaining space is distributed.

Applying Flexbox

Below is an example of how to use Flexbox to create a simple navigation bar:

HTML code:

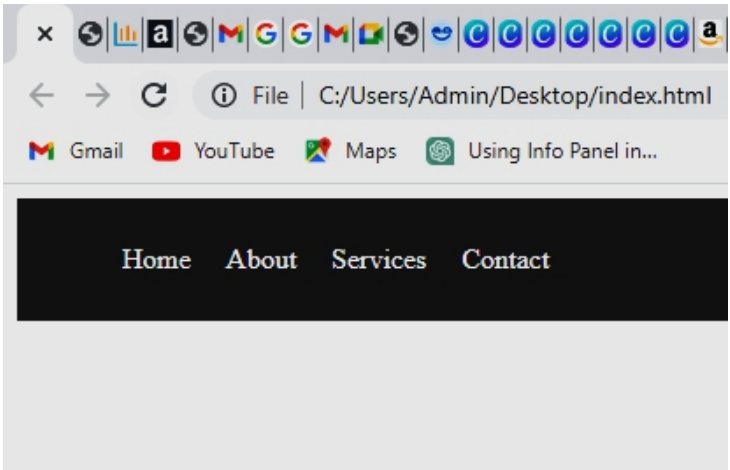
```html
<nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Services</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
```

CSS code:

```css
nav {
  display: flex;
  justify-content: space-between;
  align-items: center;
  background-color: #333;
  color: #fff;
  padding: 10px;
}

ul {
  display: flex;
  list-style: none;
}
li {
  margin: 0 10px;
}
a {
  color: #fff;
  text-decoration: none;
}
```

In this example, the nav element is set to display: flex, which turns it into a flex container. The justify-content property is set to space-between, which aligns the child elements with equal space between them. The align-items property is set to center, which vertically centers the child elements. The ul element is also set to display: flex, which turns it into a flex container, and the child li elements are given a margin to separate them. Finally, the “a” element is styled with white text color and no text decoration.

This creates a horizontal navigation bar with equal spacing between the items, centered vertically within the nav element. The navigation bar is also given a dark background color and white text color. See the result below.



CSS Variables

CSS variables, also known as CSS custom properties, are a powerful feature of modern CSS that allows you to define a variable once and use it throughout your CSS code. They are similar to variables in programming languages, in that they allow you to store and reuse values. To define a CSS variable, you can use the -- syntax followed by a name and value:

```
:root {
  --main-color: #f00;
}
```

This creates a variable named --main-color with a value of #f00. You can then use this variable in other parts of your CSS code:

```
h1 {
  color: var(--main-color);
}
```

This sets the color of all **h1** elements to the value of the **--main-**

color variable. CSS variables are useful for some reasons. For one, they allow you to define a value once and use it throughout your code, which can save you a lot of time and effort. They also allow you to easily change the value of a property across your entire website by simply updating the variable value in one place.

CSS variables can also be used in combination with JavaScript, which allows you to dynamically change the value of a variable based on user input or other events. This can be especially useful for creating dynamic and interactive websites.

Scalable Vector Graphics (SVG)

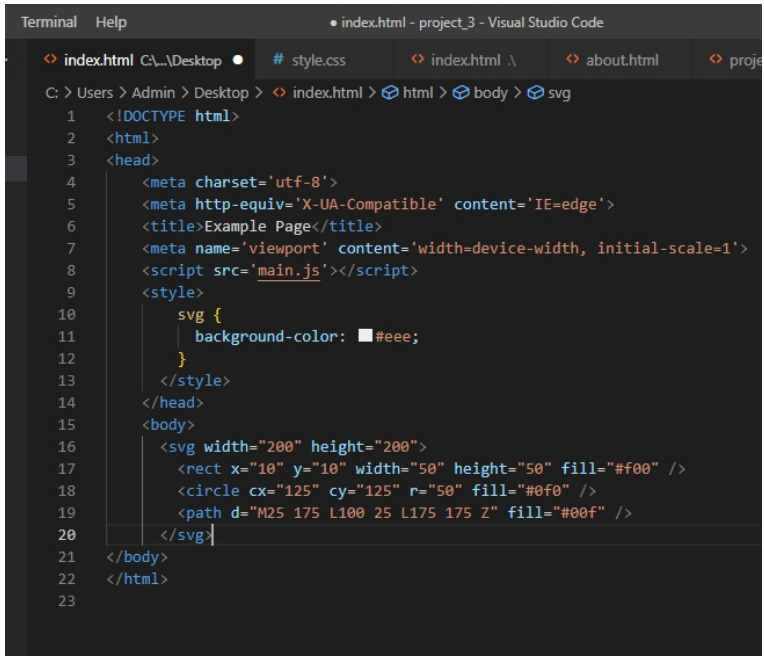
Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics that supports interactivity and animation. It was developed by the World Wide Web Consortium (W3C) and was first released in 2001. SVG images are resolution-independent, meaning they can be scaled to any size without losing their quality.

SVG images are created using shapes, lines, and curves, and can be edited using text editors or specialized software such as Adobe Illustrator. They can also be created programmatically using JavaScript.

In web development, SVG is commonly used for creating logos, icons, and other graphical elements that need to be scalable and responsive. It can also be used for creating interactive graphics and animations. SVG images can be embedded directly into HTML using the <svg> element or can be included as an external file using the <img> element or CSS background property.

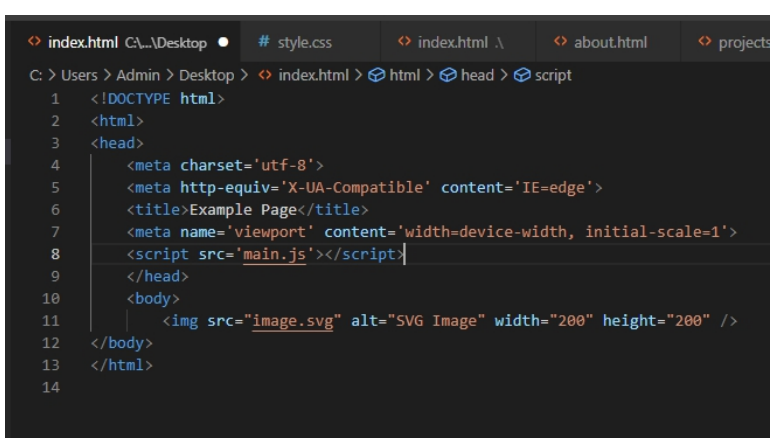
To apply Scalable Vector Graphics (SVG), you can embed SVG directly into your HTML code using the <**svg**> element, or you can include it as an external file using the <**img**> element.

To embed SVG directly, you can add the <**svg**> element to your HTML code with its required attributes, such as **width** and **height**, and then add shapes and other graphic elements inside the <**svg**> element using SVG syntax. For example:



This example creates an SVG element with a width and height of 200 pixels and a light gray background. It then adds a red square, a green circle, and a blue triangle using SVG syntax within the <**svg**> element.

To include an SVG file as an external file, you can use the <**img**> element with the **src** attribute set to the URL of the SVG file. For example:



In this example, the <**img**> element includes an SVG image with the **src** attribute set to "image.svg" and a width and height of 200 pixels.

CSS Custom Fonts

Custom fonts in CSS refer to fonts that are not commonly available on most devices but can be imported and used on a web page. Using custom fonts can add a unique touch to the design of a website and can make it stand out from other sites that use only common fonts.

To use custom fonts in CSS, you first need to download the font files and add them to your website's directory. Then, you can use the @font-face rule to import the font into your CSS. The @font-face rule specifies the font family name and the URL of the font file, and can also include other properties such as font-weight and font-style.

Here is an example of how to use the @font-face rule to import a custom font:

```
@font-face {
  font-family: 'MyCustomFont';
  src: url('mycustomfont.woff2') format('woff2'),
       url('mycustomfont.woff') format('woff');
  font-weight: 400;
  font-style: normal;
}

body {
  font-family: 'MyCustomFont', sans-serif;
}
```

In this example, the font family name is set to 'MyCustomFont', and the font files are located in the same directory as the CSS file. The font-weight is set to 400 (normal) and the font-style is set to normal. Finally, the font family is applied to the body element, with the fallback font being a sans-serif font.

CSS Best Practices

CSS Best Practices are a set of guidelines or recommendations for writing clean, maintainable, and efficient CSS code. They help developers to write better code, avoid common mistakes, and make

their code more scalable and reusable.

Some common CSS best practices include:

- **Use meaningful and consistent class names**: Class names should accurately describe the content they represent and be consistent throughout the project.
- **Avoid using inline styles**: Inline styles make it difficult to maintain and update code, as they are scattered throughout the HTML markup.
- **Use a consistent indentation style**: Use consistent indentation to improve readability and make it easier to identify the structure of your code.
- **Use shorthand properties when possible**: Shorthand properties allow you to write more concise code and reduce the overall file size.
- **Keep your code organized**: Organize your code into separate files, and group related styles together.
- **Avoid using excessive specificity**: Excessive specificity makes it difficult to override styles and can lead to specificity wars.
- **Use comments to document your code**: Comments can be used to describe the purpose and function of your code and make it easier to maintain.
- **Minimize the use of !important**: The !important rule should be used sparingly, as it can make it difficult to override styles.
- **Use a CSS preprocessor**: CSS preprocessors, such as Sass or Less, can help you write more efficient and maintainable CSS code.
- **Test your code in multiple browsers**: Testing your code in multiple browsers helps to ensure cross-browser compatibility and avoid unexpected rendering issues.

Writing Efficient CSS

Writing efficient CSS is an important skill to ensure that your web pages load quickly and smoothly. Here are some best practices for writing efficient CSS:

- **Minimize the use of selectors**: The more complex your

CSS selectors are, the slower the browser will be able to render the page. Try to use the simplest selectors possible and avoid using too many descendant selectors.

- **Use shorthand properties**: Instead of using separate properties for margin, padding, and borders, use the shorthand properties. This reduces the amount of CSS you need to write and can also make your code easier to read.
- **Avoid using too many fonts**: Using too many different fonts on a page can slow down page load time. Try to limit yourself to using two or three fonts.
- **Use CSS resets**: Browsers have their default styles for HTML elements, which can cause inconsistencies in how your web pages look. Using a CSS reset like Normalize.css can help ensure that your web pages look consistent across different browsers.
- **Group-related CSS properties**: When writing CSS, group related properties together. For example, instead of writing separate rules for margin-top, margin-bottom, margin-left, and margin-right, group them in a margin rule.
- **Use efficient CSS animations**: When using CSS animations, avoid using too many keyframes or animating too many properties at once. This can slow down page load time and make your animations less smooth.

Debugging CSS

Debugging CSS is an important part of web development. Here are some tips for debugging CSS:

- **Use the browser's built-in developer tools**: Most modern web browsers come with built-in developer tools that allow you to inspect and debug your CSS. You can use the elements tab to view the HTML and CSS of your web page, and the console tab to view any errors or warnings.
- **Check for typos and syntax errors**: CSS is very sensitive to typos and syntax errors. Check your CSS code for any typos, missing semicolons, or other syntax

errors.

- **Use meaningful class and ID names**: Use class and ID names that are descriptive and meaningful. This will make it easier to understand your CSS code and debug any issues.
- **Use comments to organize your code**: Use comments to organize your CSS code and make it easier to understand. This can also help you identify any issues or bugs in your code.
- **Use a CSS preprocessor**: CSS preprocessors like Sass or Less can help you write more efficient and organized CSS code. They also offer features like variables and mixins that can help you avoid repeating code.
- **Use a CSS linter**: A CSS linter can help you identify and fix common CSS errors and improve the overall quality of your code.
- **Test your code in different browsers**: CSS can behave differently in different browsers, so it's important to test your code in multiple browsers to ensure it works as expected.

The Future of CSS

The future of CSS is an exciting prospect, as new features and capabilities are constantly being developed and implemented. Here are some of the trends and developments that are shaping the future of CSS:

- **CSS Grid:** CSS Grid is a powerful layout system that allows for complex layouts to be created quickly and easily. It has already gained widespread adoption and is becoming the standard for layout design.
- **CSS Custom Properties**: CSS custom properties, or variables, allow developers to define reusable values and apply them to various parts of their CSS, making it easier to maintain and update styles.
- **CSS Animations and Transitions**: CSS animations and transitions are becoming more sophisticated, with new properties and features being added that allow for more complex and interactive animations.
- **Responsive Web Design**: Responsive design has

already become the standard for web design, but as more devices and screen sizes emerge, the need for responsive design will only continue to grow.

- **Accessibility**: Accessibility is becoming increasingly important, and CSS plays a vital role in making websites accessible to all users. New CSS features and techniques are being developed to make it easier to create accessible designs.

CHAPTER NINE

HTML AND CSS PRACTICAL EXERCISES

This chapter consists of HTML and CSS Practical Exercises that are intended to boost beginners and advanced web developer 'ability in building professional websites with HTML and CSS.

Exercise 1: Design a simple login page with HTML and CSS

Below is an example of HTML and CSS code for a simple Login page.

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Page</title>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <div class="login-container">
    <h1>Login</h1>
    <form>
      <label for="username">Username</label>
      <input type="text" id="username" name="username">
      <label for="password">Password</label>
      <input type="password" id="password" name="password">
      <button type="submit">Sign In</button>
    </form>
  </div>
</body>
</html>
```

CSS Code:

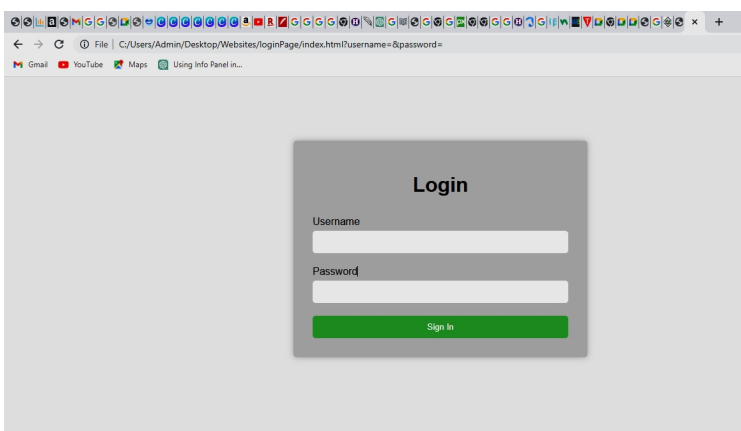
Create a CSS file and include the following code.

```
body {
  margin: 0;
  padding: 0;
  font-family: Arial, sans-serif;
```

```
  background-color: #f2f2f2;
}
.login-container {
  width: 400px;
  margin: 100px auto;
  background-color: #bbbbbb;
  padding: 30px;
  border-radius: 5px;
  box-shadow: 0px 0px 10px 0px rgba(0,0,0,0.3);
}
h1 {
  text-align: center;
  margin-bottom: 30px;
}
form {
  display: flex;
  flex-direction: column;
}
label {
  margin-bottom: 5px;
}
input {
  padding: 10px;
  margin-bottom: 20px;
  border: none;
  border-radius: 5px;
```

```
}
button {
  background-color: #4CAF50;
  color: white;
  padding: 10px;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}
button:hover {
  background-color: #3e8e41;
}
```

The web result is shown below.



Exercise 2: Build a simple landing page for a product with HTML and CSS.

Below is an example of HTML and CSS code for a simple landing page for a product:

HTML code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <title>Product Landing Page</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Features</a></li>
        <li><a href="#">Pricing</a></li>
        <li><a href="#">Contact</a></li>
      </ul>
    </nav>
    <h1>Mouse</h1>
  </header>
  <main>
    <section>
      <h2>Features</h2>
      <ul>
```

```
        <li>Rechageable</li>
        <li>8hrs Battery Life</li>
        <li>Grey Color</li>
      </ul>
    </section>
    <section>
      <h2>Pricing</h2>
      <p>Price: $20</p>
      <button>Buy Now</button>
    </section>
  </main>
  <footer>
    <p>Copyright; 2023 Sammie Smith. All rights reserved.</p>
  </footer>
</body>
</html>
```

CSS code:

Create a CSS file and include the following code.

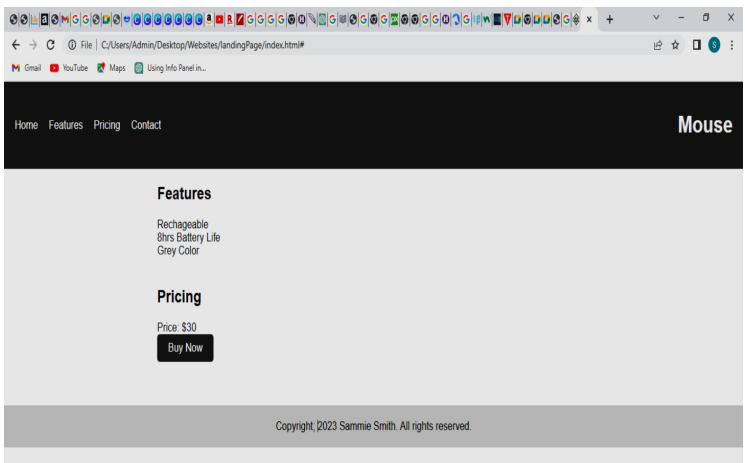
```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
  }
    header {
    background-color: #333;
    color: white;
```

```
  padding: 20px;
  display: flex;
  justify-content: space-between;
  align-items: center;
}
  nav ul {
  list-style: none;
  margin: 0;
  padding: 0;
  display: flex;
}
  nav li {
  margin-right: 20px;
}
  nav li:last-child {
  margin-right: 0;
}
  nav a {
  color: white;
  text-decoration: none;
}
  main {
  padding: 20px;
  max-width: 800px;
  margin: 0 auto;
}
```

```
  section {
  margin-bottom: 40px;
}
  h2 {
  margin-top: 0;
}
ul {
  list-style: none;
  margin: 0;
  padding: 0;
}
p {
  margin: 0;
}
button {
  background-color: #333;
  color: white;
  border: none;
  padding: 10px 20px;
  border-radius: 5px;
  cursor: pointer;
  font-size: 16px;
}
button:hover {
  background-color: #666;
}
```

```
footer {
    background-color: #ccc;
    padding: 20px;
    text-align: center;
}
```

This code creates a simple landing page with a navigation bar, product name, features section, pricing section, and footer. You can customize the content and styling to fit your product's needs. See the web result below.



Note that you can modify the exercise above to suite your project.

Exercise 3: Create a responsive pricing table with HTML and CSS

Below is an example of a responsive pricing table using HTML and CSS:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Pricing Table</title>
    <link rel="stylesheet" type="text/css" href="style.css">
  </head>
  <body>
    <div class="pricing-table">
      <div class="plan">
        <h2>Basic</h2>
        <h3>$10/month</h3>
        <ul>
          <li>1 User</li>
          <li>5GB Storage</li>
          <li>Unlimited Projects</li>
          <li>Free Support</li>
        </ul>
        <a href="#" class="button">Choose Plan</a>
      </div>

      <div class="plan popular">
        <h2>Pro</h2>
        <h3>$20/month</h3>
        <ul>
          <li>5 Users</li>
          <li>20GB Storage</li>
          <li>Unlimited Projects</li>
```

```
        <li>Free Support</li>
      </ul>
      <a href="#" class="button">Choose Plan</a>
    </div>
    <div class="plan">
      <h2>Advanced</h2>
      <h3>$30/month</h3>
      <ul>
        <li>Unlimited Users</li>
        <li>50GB Storage</li>
        <li>Unlimited Projects</li>
        <li>Free Support</li>
      </ul>
      <a href="#" class="button">Choose Plan</a>
    </div>
  </div>
</body>
</html>
```

CSS Code:

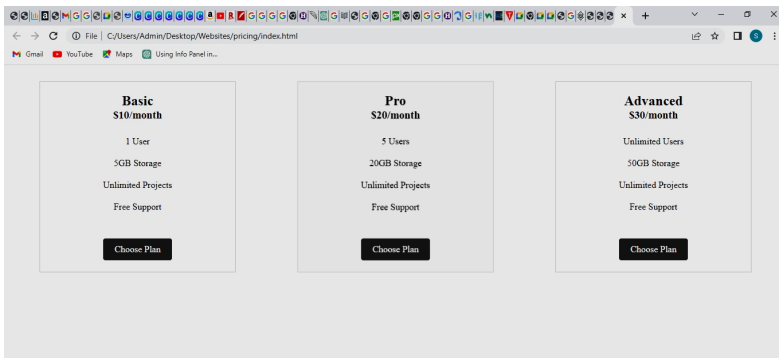
Create a CSS file and include the following code.

```
.pricing-table {
    display: flex;
    justify-content: space-around;
    flex-wrap: wrap;
  }
  .plan {
```

```
  text-align: center;
  margin: 20px;
  padding: 20px;
  border: 1px solid #ccc;
  width: 30%;
  max-width: 300px;
  transition: all 0.3s ease-in-out;
}
.plan:hover {
  transform: scale(1.1);
  box-shadow: 0 5px 10px rgba(0,0,0,0.3);
}
.popular {
  background-color: #f9f9f9;
}
h2, h3 {
  margin: 0;
}
ul {
  list-style: none;
  padding: 0;
}
li {
  padding: 10px 0;
}
.button {
```

```
    display: inline-block;
    padding: 10px 20px;
    margin-top: 20px;
    background-color: #333;
    color: #fff;
    text-decoration: none;
    border-radius: 4px;
}
```

The web result is shown as follows.



CHAPTER TEN

RESPONSIVE WEB DESIGN

Responsive web design is an approach to web design that focuses on creating websites that are optimized for a variety of devices and screen sizes. With the increasing popularity of mobile devices, responsive web design has become essential for ensuring that websites are accessible and user-friendly on all devices.

Responsive web design involves using CSS media queries to adjust the layout and content of a website based on the size of the screen it is being viewed on. This allows for a seamless user experience, regardless of the device being used to access the website.

In this chapter, we will explore the principles of responsive web design, including best practices for designing responsive websites and the tools and technologies used to create them. We will also cover some common challenges and solutions associated with responsive web design, such as optimizing images and managing complex layouts.

Introduction to Responsive Design

Responsive design is an approach to web design that ensures websites and applications look good and function well across multiple devices and screen sizes. With the growing use of mobile devices to access the internet, responsive design has become a critical aspect of modern web development.

Responsive design typically involves using a combination of techniques, such as fluid layouts, flexible images, and media queries, to create websites that can adapt to different screen sizes and resolutions. The goal of responsive design is to provide a seamless and consistent user experience, regardless of the device being used to access the website. This means that elements on the page will reposition, resize, or even disappear based on the screen size and orientation, without losing any functionality or readability.

Responsive design ensures that websites and applications are accessible to all users, regardless of the device they are using and helps to ensure that they can navigate and interact with the content in the most efficient and user-friendly way possible.

Best Practices for Designing Responsive Websites

Below are some best practices for designing responsive websites:

- **Mobile-first approach**: Start by designing for the smallest screen size first and then gradually move towards larger screen sizes. This ensures that the website is optimized for smaller screens and loads faster.
- **Use a responsive framework**: Responsive frameworks like Bootstrap provide a grid system that helps in creating responsive layouts quickly. This also ensures

that the website is compatible with different screen sizes.

- **Optimize images**: Large images can slow down the website's loading time, so it's important to optimize them for web use. You can use image compression tools like TinyPNG or JPEGmini to reduce the size of images without affecting their quality.
- **Use media queries:** Media queries allow you to apply different styles to different screen sizes. This helps in creating a consistent user experience across different devices.
- **Test on multiple devices**: It's important to test the website on different devices to ensure that it looks and functions correctly on all of them. You can use tools like BrowserStack or Device Mode in Google Chrome to test the website on different devices.
- **Keep it simple**: Avoid using too many complex design elements or animations that can slow down the website's loading time. Stick to simple and clean design principles that are easy to read and navigate on smaller screens.
- **Use scalable fonts**: Use fonts that are scalable and easy to read on smaller screens. Avoid using fonts that are too small or difficult to read on smaller screens.

Tools and Technologies used to Create Responsive Website

There are several tools and technologies used to create responsive websites. Some of the most popular ones include:

- **HTML and CSS**: These are the basic building blocks of any website and are used to create the structure and style of the website.
- **CSS frameworks**: Frameworks like Bootstrap and Foundation provide pre-built CSS styles and components that make it easier to create responsive websites.
- **CSS preprocessors**: Preprocessors like Sass and Less allow developers to write CSS more efficiently by

providing features like variables, nesting, and mixins.

- **JavaScript**: JavaScript can be used to add interactivity and dynamic behavior to a website. It is often used for things like responsive navigation menus and sliders.
- **Responsive design tools**: There are many tools available that help with responsive design, such as media query generators, viewport testing tools, and responsive design frameworks.
- **Content management systems (CMS):** CMS platforms like WordPress, Drupal, and Joomla have built-in responsive design features and templates that can be used to create responsive websites.
- **Mobile app development platforms**: Platforms like React Native and Flutter allow developers to create mobile apps that are optimized for different devices and screen sizes.

Understanding Media Queries

Media queries are a CSS technique used for styling web pages based on the device or screen size they are viewed on. With media queries, you can define different styles for different devices or screen sizes, such as desktops, laptops, tablets, and mobile devices.

Media queries work by checking the width and height of the viewport, which is the area of the browser window where the web page is displayed. You can set a specific style for a particular viewport size or range of sizes by using the **@media** rule in CSS.

For example, you could define a media query for screens with a maximum width of 768 pixels and set a different font size and layout for that size range. This allows you to create a more user-friendly and consistent experience across different devices.

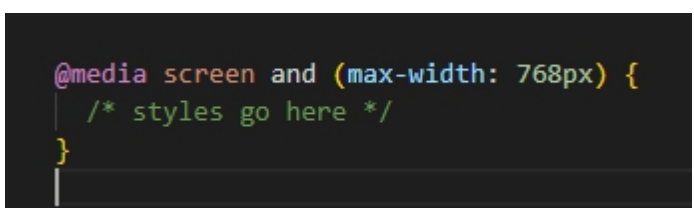
Media queries can also be used to apply different styles based on other characteristics of the device, such as orientation (landscape or portrait), resolution, or color depth.

Applying Media Queries

To apply media queries in CSS, you first need to specify the media type you want to apply the styles to. The most common media type

is the screen, which is used for devices with a screen, such as desktop computers, laptops, tablets, and smartphones. Other media types include print, which is used for printing, and speech, which is used for screen readers. To apply styles to a specific media type, you can use the @media rule followed by the media type in parentheses, like this:

Within the media query block, you can write styles that will only apply when the specified media type is in use. For example, you can set different font sizes, colors, or layout properties for desktop and mobile devices. You can also use media queries to specify different styles based on the width or height of the device's screen. For example, you can write styles that will only apply when the screen width is less than or equal to 768 pixels:



This will apply the styles within the media query block when the screen width is 768 pixels or less. You can also use the **min-width** property to specify a minimum screen width for the styles to apply.

By using media queries, you can create responsive designs that adjust to different screen sizes and devices, making your website more user-friendly and accessible.

Understanding Mobile-first Design Approach

Mobile-first design is a design approach in which a website or application is designed for mobile devices first and then scaled up to fit larger screens such as desktops or laptops. This approach

assumes that more people access websites through their mobile devices than their desktops, and it aims to create an optimal experience for mobile users. When designing for mobile first, the focus is on creating a streamlined, efficient user experience that prioritizes the most important content and tasks. This often involves using a minimalist design, simplifying navigation, and using large, easy-to-read fonts and buttons.

Once the mobile design is in place, the design can be scaled up to fit larger screens. This involves using media queries to adjust the layout and styling of the website based on the screen size of the device being used. Mobile-first design has become increasingly important as mobile devices have become the primary way that people access the internet. By designing with mobile in mind, developers can ensure that their websites are accessible and usable for the majority of their audience.

Applying Mobile-first Design

To carry out a mobile-first design using CSS, follow these steps:

- Start with a mobile-first CSS file, which contains styles for the smallest screen sizes first, and then gradually adds styles for larger screen sizes as necessary. For example:

```css
/* Styles for mobile devices */
@media (min-width: 320px) {
  /* Styles for screens larger than 320px */
}

@media (min-width: 480px) {
  /* Styles for screens larger than 480px */
}

@media (min-width: 768px) {
  /* Styles for screens larger than 768px */
}

@media (min-width: 1024px) {
  /* Styles for screens larger than 1024px */
}
```

- Use responsive units such as **em** and **%** instead of fixed units like **px** to ensure that elements scale properly on different screen sizes.
- Use a responsive framework such as Bootstrap or Foundation, which provide pre-built CSS classes and components for creating responsive layouts.
- Use CSS media queries to target specific screen sizes and apply custom styles as needed.
- Test your design on various devices and screen sizes to ensure that it looks good and functions properly across all platforms.

Study the code below to understand how to implement Mobile-first Design.

HTML Code:

<!DOCTYPE html>

<html>

<head>

<meta charset='utf-8'>

```
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Example Page</title>
 <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' type='text/css' media='screen' href='main.css'>
    <script src='main.js'></script>
</head>
<body>
<header>
<h1>My Website</h1>
<nav>
<ul>
<li><a href="#">Home</a></li>
<li><a href="#">About</a></li>
<li><a href="#">Contact</a></li>
</ul>
</nav>
</header>
<main>
<section>
<h2>About Me</h2>
<p>Hi, my name is John Doe and I am a web developer.</p>
</section>
<section>
<h2>Recent Work</h2>
<div class="work">
```

<img src="work1.jpg" alt="Work 1">

<img src="work2.jpg" alt="Work 2">

<img src="work3.jpg" alt="Work 3">

* *

</section>

</main>

<footer>

p> Copyrights @ 2023 My Website. All Rights Reserved.</p>

</footer>

</body>

</html>

Afterward, create a CSS external file and link it to your html file and include the code below. Ensure to use information suitable to the project you are working on.

CSS Code:

** {*

box-sizing: border-box;

}

body {

font-family: Arial, sans-serif;

font-size: 16px;

margin: 0;

padding: 0;

}

header {

background-color: #333;

color: #fff;

```
padding: 10px;
}
nav ul {
list-style: none;
margin: 0;
padding: 0;
display: flex;
flex-wrap: wrap;
}
nav li {
margin: 0 10px;
}
nav a {
color: #fff;
text-decoration: none;
}
main {
padding: 20px;
}
section {
margin-bottom: 20px;
}
section h2 {
font-size: 24px;
margin-bottom: 10px;
}
```

```
section p {
font-size: 16px;
line-height: 1.5;
}
.work {
display: flex;
flex-wrap: wrap;
}
.work a {
flex: 1 1 300px;
margin: 10px;
}
@media screen and (min-width: 768px) {
header {
display: flex;
justify-content: space-between;
align-items: center;
}
nav {
flex: 1;
}
main {
display: flex;
flex-wrap: wrap;
}
section {
```

flex: 1;

margin: 0 10px;

}

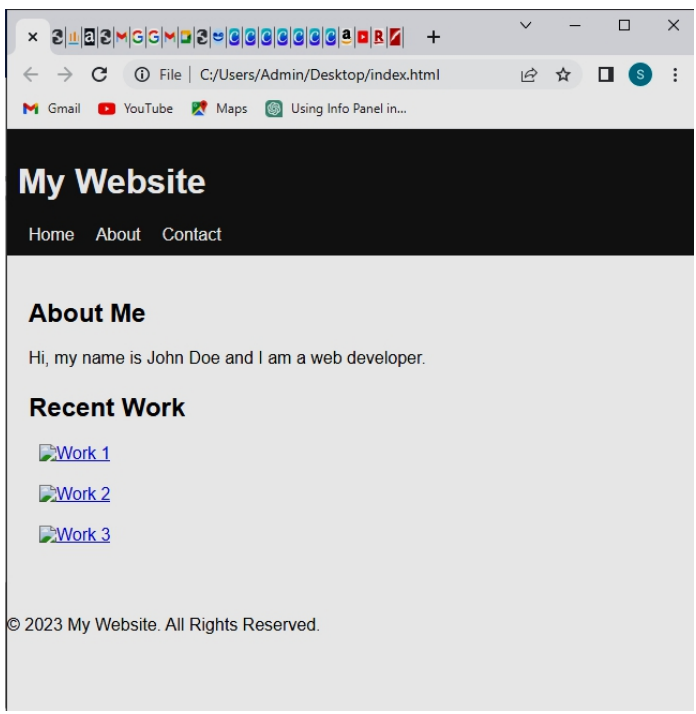
.work a {

margin: 10px 0;

}

}

This code includes a simple header, navigation, main content with two sections, and a footer. The CSS includes styles for mobile devices with a small screen width of fewer than 768 pixels, as well as larger screen sizes with a media query. When the screen width is at least 768 pixels, the header, and main content are displayed using a flexbox layout. See the result below.



Note that you can always modify the codes in this guide to suit your taste.

Responsive Frameworks

Responsive frameworks are pre-built collections of CSS and JavaScript files that provide a foundation for creating responsive web designs. They typically include a grid system and pre-defined styles for common HTML elements, as well as pre-built components like navigation menus, buttons, and forms.

Using a responsive framework can greatly simplify the process of building responsive websites, as it eliminates the need to write custom CSS for every design element. Instead, developers can use the framework's pre-built classes and components to quickly create responsive layouts and designs. Some popular responsive frameworks include Bootstrap, Foundation, and Bulma. These frameworks are widely used and well-documented, making them a great choice for both beginner and experienced developers.

Note that to make a completely interactive website with a powerful feature you would require all the frameworks put together, but this book is limited to HTML and CSS.

Common Challenges and Solutions Associated with Responsive Web Design

Responsive web design can present some challenges that need to be addressed to provide a seamless experience across all devices. Some common challenges and solutions associated with responsive web design include:

- **Device compatibility**: With the wide variety of devices available today, ensuring that your website looks good on all of them can be a challenge. The solution is to use responsive design techniques that adapt to different screen sizes, such as flexible grids, responsive images, and media queries.
- **Content prioritization**: When designing for smaller screens, it's important to prioritize content so that the most important information is displayed first. One solution is to use a mobile-first approach, where you design for the smallest screen size first and then add more content and features as the screen size increases.

- **Navigation**: Navigation menus can be tricky on small screens, as they can take up valuable screen real estate. A solution is to use a hamburger menu, where the menu is hidden behind a small icon until the user taps on it.
- **Performance**: Large images and complex layouts can slow down the performance of your website, especially on mobile devices with slower internet connections. A solution is to optimize images and use techniques such as lazy loading to improve page speed.
- **Testing**: Testing your website on different devices and screen sizes can be time-consuming and difficult. A solution is to use responsive design testing tools that allow you to preview your website on different devices and screen sizes, such as BrowserStack, Responsinator, or Am I Responsive.

Optimizing Images and Managing Complex Layouts

As part of the process of building responsive websites, optimizing images and managing complex layouts are two critical areas to focus on. Optimizing images involves reducing the file size of images to improve website performance and load times. This can be achieved through techniques such as compressing images, resizing images to the appropriate dimensions, and using the appropriate image file format. There are also various tools available to help optimize images, such as Photoshop and online tools like TinyPNG.

Managing complex layouts involves creating layouts that work across a variety of screen sizes and devices. This can be achieved through the use of CSS grid systems and frameworks, which provide pre-defined grids and responsive design patterns that can be easily adapted to different screen sizes. Additionally, designing with a mobile-first approach can help ensure that the most important content is prioritized for smaller screens, with additional content and layout elements added as the screen size increases.

Other common challenges in responsive web design include issues with typography, navigation, and performance optimization. However, with careful planning and attention to detail, these

challenges can be overcome, resulting in a well-designed and functional responsive website.

CHAPTER ELEVEN

RESPONSIVE WEBSITES PRACTICAL EXERCISES

In this chapter, we will focus on practical exercises that will help you create a responsive website from scratch. You will learn how to use HTML and CSS to build a website that looks good on desktop computers, tablets, and smartphones.

We will cover various topics such as layout design, media queries, and responsive images. By the end of this chapter, you will be able to create a website that looks great on any device, making it accessible to a wider audience.

Exercise 1: Design a responsive restaurant website

Design a responsive restaurant website with a homepage, menu page, reservation page, and contact page. Use responsive design techniques to ensure the website looks good on all devices and provides a user-friendly experience for customers with HTML and

CSS.

Below is an example of how you can create a restaurant website.

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Restaurant Homepage</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Menu</a></li>
        <li><a href="#">Reservation</a></li>
        <li><a href="#">Contact</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section class="hero">
```

```
  <h1>Welcome to Our Restaurant</h1>
  <p>Enjoy delicious food and great ambiance</p>
  <a href="#">See Menu</a>
</section>
<section class="menu">
  <h2>Our Menu</h2>
  <ul>
    <li>
      <img src="img1.jpeg" alt="Pizza">
      <h3>Pizza</h3> <br>
      <p>Delicious pizza with fresh toppings</p>
      <span>$10</span>
    </li>
    <li>
      <img src="img2.jpeg" alt="Burger">
      <h3>Burger</h3>
      <p>Classic burger with all the fixings</p>
      <span>$8</span>
    </li>
    <li>
      <img src="img3.jpeg" alt="Pasta">
      <h3>Pasta</h3>
      <p>Freshly made pasta with delicious sauce</p>
      <span>$12</span>
    </li>
  </ul>
```

```
</section>
<section class="reservation">
  <h2>Make a Reservation</h2>
  <form>
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email">
    <label for="date">Date:</label>
    <input type="date" id="date" name="date">
    <label for="time">Time:</label>
    <input type="time" id="time" name="time">
    <label for="guests">Number of Guests:</label>
    <input type="number" id="guests" name="guests">
    <button type="submit">Submit</button>
  </form>
</section>
<section class="contact">
  <h2>Contact Us</h2>
  <form>
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email">
    <label for="message">Message:</label>
    <textarea id="message" name="message"></textarea>
```

```
        <button type="submit">Submit</button>
      </form>
    </section>
  </main>
  <footer>
    <p>&copy; 2023 Yummies Restaurant</p>
  </footer>
</body>
</html>
```

CSS Code:

```
/* Global styles */
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
  }
    body {
    font-family: Arial, sans-serif;
  }
    header {
    background-color: #333;
    color: #fff;
    padding: 10px;
  }
    nav ul {
    list-style: none;
```

```
  margin: 0;
  padding: 0;
}
  nav li {
  display: inline-block;
  margin-right: 20px;
}
  nav a {
  color: #fff;
  text-decoration: none;
}
  main {
  padding: 20px;
}
  section {
  margin-bottom: 40px;
}
  h1 {
  font-size: 48px;
  margin-bottom: 20px;
}
  h2 {
  font-size: 36px;
  margin-bottom: 10px;
}
  ul {
```

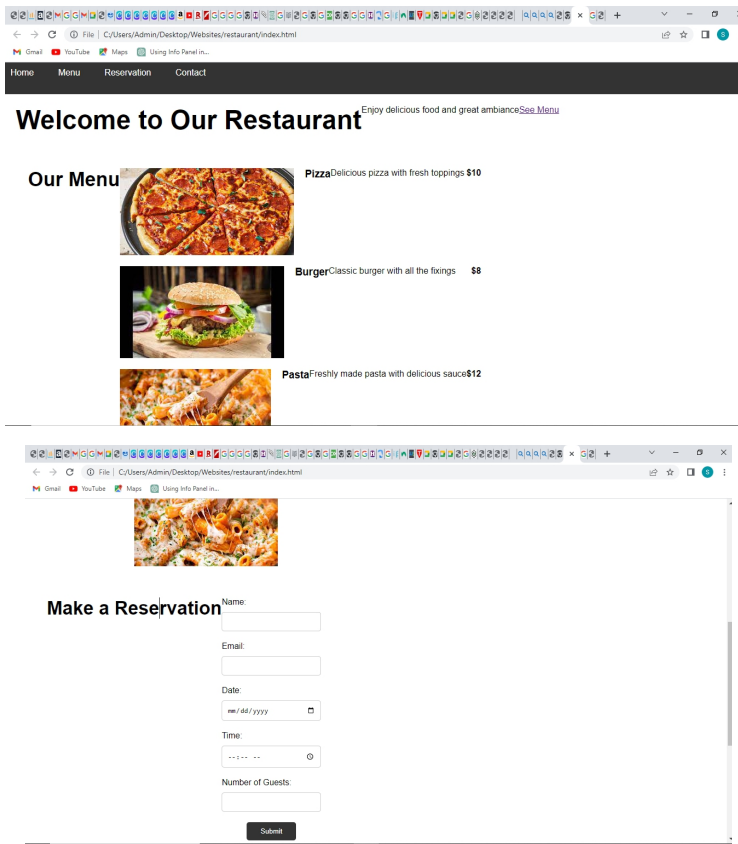
```
  list-style: none;
  margin: 0;
  padding: 0;
}
  li {
  display: flex;
  margin-bottom: 20px;
}
  img {
  margin-right: 20px;
}
  span {
  margin-left: auto;
  font-weight: bold;
}
  form {
  display: flex;
  flex-direction: column;
}
  label {
  margin-bottom: 10px;
}
  input,
textarea,
button {
  padding: 10px;
```

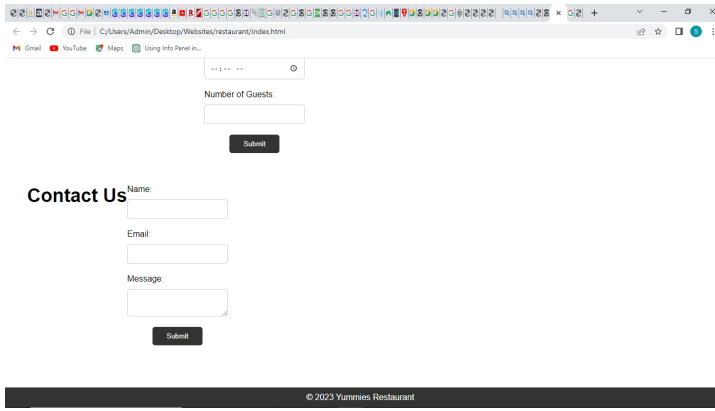
```
  margin-bottom: 20px;
  border: 1px solid #ccc;
  border-radius: 5px;
}
  input:focus,
textarea:focus {
  outline: none;
  border-color: #333;
}
  button {
  background-color: #333;
  color: #fff;
  border: none;
  cursor: pointer;
}
  button:hover {
  background-color: #555;
}
  footer {
  background-color: #333;
  color: #fff;
  text-align: center;
  padding: 10px;
}
  /* Media queries */
  @media (min-width: 768px) {
```

```
/* Header */
nav li {
  margin-right: 40px;
}
  /* Sections */
section {
  display: flex;
  flex-wrap: wrap;
}
  .hero {
  flex-basis: 50%;
  padding-right: 40px;
}
  .menu,
.reservation,
.contact {
  flex-basis: 27%;
  padding-left: 22px;
  padding-right: 22px;
}
  /* Form fields */
label,
input,
textarea,
button {
  width: 100%;
```

```
  }
    /* Form buttons */
  button {
    width: 50%;
    margin-left: auto;
    margin-right: auto;
  }
}
```

See the web results below.





Exercise 2: Create a Responsive Photographer Portfolio

Create a responsive online portfolio for a photographer, artist, or designer. Showcase their work in a visually appealing way, using responsive design techniques to ensure the website looks good on all devices with HTML and CSS.

Below is an example of HTML and CSS code to create a responsive online portfolio for a photographer:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Photographer Portfolio</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
```

```
<header>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">Portfolio</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
</header>
<main>
  <section class="hero">
    <h1>Welcome to My Portfolio</h1>
    <p>Check out my latest work</p>
  </section>
  <section class="portfolio">
    <h2>My Portfolio</h2>
    <ul>
      <li>
        <img src="img7.jpg" alt="Photograph 1">
        <h3>Photograph 1</h3>
      </li>
      <li>
        <img src="img5.jpeg" alt="Photograph 2">
        <h3>Photograph 2</h3>
      </li>
```

```
        <li>
          <img src="img6.jpeg" alt="Photograph 3">
          <h3>Photograph 3</h3>
        </li>
        <li>
          <img src="img4.jpeg" alt="Photograph 4">
          <h3>Photograph 4</h3>
        </li>
      </ul>
    </section>
    <section class="about">
      <h2>About Me</h2>
      <p>Hi, I'm a photographer based in New York City. I specialize in landscape and portrait photography. My goal is to capture the beauty of the world and tell stories through my photographs.</p>
    </section>
    <section class="contact">
      <h2>Contact Me</h2>
      <form>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email">
        <label for="message">Message:</label>
        <textarea id="message" name="message"></textarea>
        <button type="submit">Submit</button>
      </form>
```

```
    </section>
  </main>
  <footer>
    <p>&copy; 2023 Sammie Smith</p>
  </footer>
</body>
</html>
```

CSS Code:

```
/* Global Styles */
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}
  body {
  font-family: Arial, sans-serif;
  font-size: 16px;
  line-height: 1.5;
  color: #130101;
}
  a {
  text-decoration: none;
  color: #4b0b0b;
}
  ul {
  list-style: none;
```

```
}
  img {
  max-width: 100%;
  height: auto;
}
  /* Header Styles */
  header {
  background-color: #7c7a7a;
  border-bottom: 1px solid #696262;
}
  nav ul {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin: 0;
  padding: 1rem;
}
  nav li {
  margin: 0 1rem;
}
  nav a {
  font-weight: bold;
  letter-spacing: 1px;
  text-transform: uppercase;
}
  /* Hero Section Styles */
```

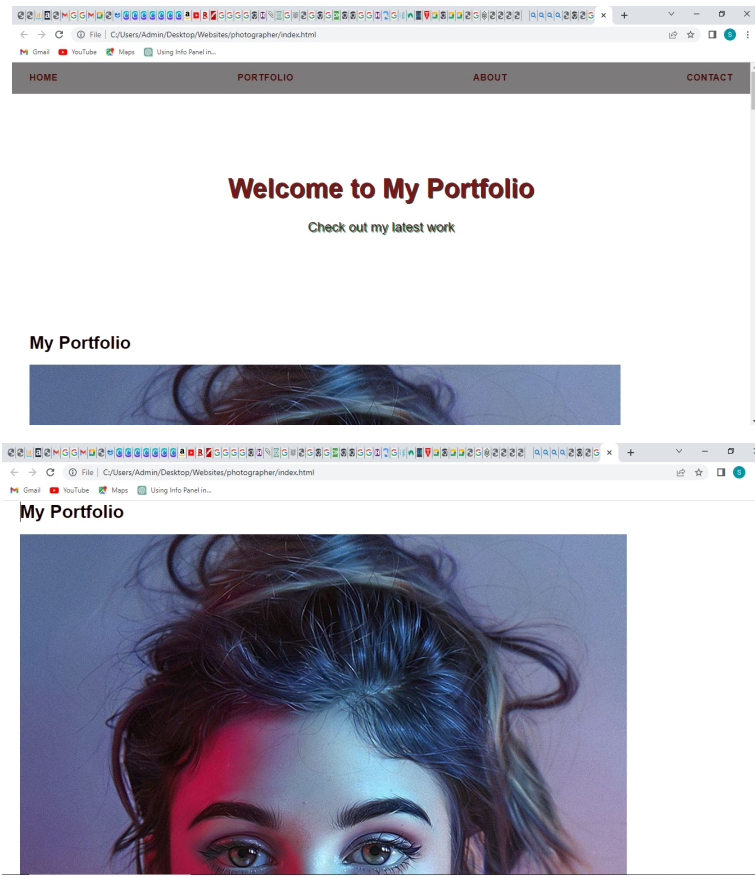
```
  .hero {
  background-image: url(https://via.placeholder.com/1200x600);
  background-size: cover;
  background-position: center;
  height: 60vh;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  text-align: center;
}
  .hero h1 {
  font-size: 3rem;
  margin-bottom: 1rem;
  color: #791818;
  text-shadow: 2px 2px #333;
}
  .hero p {
  font-size: 1.5rem;
  color: #442222;
  text-shadow: 2px 2px #418d58;
}
  /* Portfolio Section Styles */
  .portfolio {
  padding: 2rem;
}
```

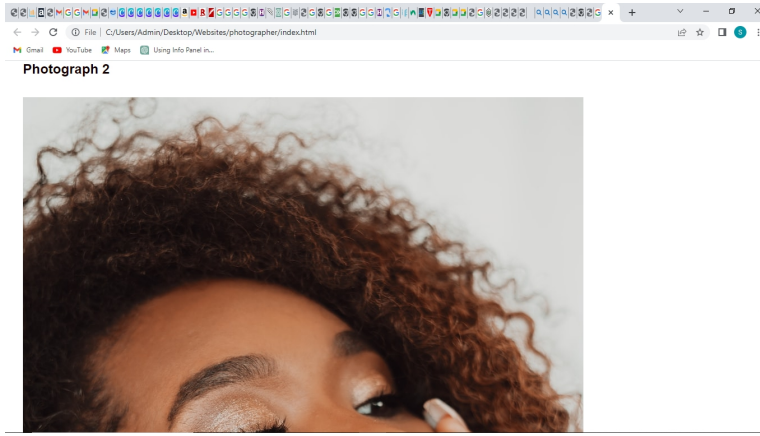
```
  .portfolio h2 {
  font-size: 2rem;
  margin-bottom: 1rem;
}
  .portfolio li {
  margin-bottom: 2rem;
}
  .portfolio h3 {
  font-size: 1.5rem;
  margin-top: 1rem;
}
  /* About Section Styles */
  .about {
  padding: 2rem;
}
  .about h2 {
  font-size: 2rem;
  margin-bottom: 1rem;
}
  /* Contact Section Styles */
  .contact {
  padding: 2rem;
}
.contact h2 {
  font-size: 2rem;
  margin-bottom: 1rem;
```

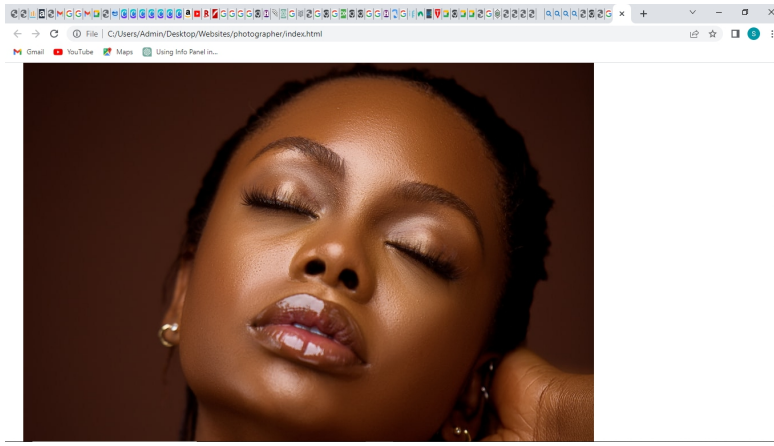
```
}
label {
  display: block;
  margin-bottom: 0.5rem;
}
 input,
textarea {
  display: block;
  width: 100%;
  padding: 0.5rem;
  margin-bottom: 1rem;
  border: 1px solid #ccc;
  border-radius: 4px;
}
 button[type="submit"] {
  background-color: #333;
  color: #fff;
  border: none;
  padding: 0.5rem 1rem;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1rem;
  text-transform: uppercase;
}
 button[type="submit"]:hover {
  background-color: #444;
```

```
}
 /* Footer Styles */
  footer {
  background-color: #333;
  color: #fff;
  text-align: center;
  padding: 1rem;
}
```

See the web result below.







Exercise 3: Develop a Responsive E-commerce Website

Develop a responsive e-commerce website with a homepage, product listing page, and product details page. Use responsive design techniques to ensure the website looks good on all devices and provides a seamless shopping experience with HTML and CSS.

Below is an example of a responsive e-commerce website.

HTML Code:

<!DOCTYPE html>

<html lang="en">

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Responsive E-commerce Website</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Shop</a></li>
        <li><a href="#">Contact</a></li>
        <li><a href="#">Cart</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section class="hero">
      <h1>Welcome to our E-commerce Website</h1>
      <p>Shop the latest products now</p>
      <a href="#" class="btn">Shop Now</a>
    </section>
    <section class="featured-products">
```

```
    <h2>Featured Products</h2>
    <ul>
      <li>
        <a href="#">
          <img src="img1.jpg" alt="Product 1">
          <h3>Product 1</h3>
          <p>$20.00</p>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img3.jpg" alt="Product 2">
          <h3>Product 2</h3>
          <p>$25.00</p>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img4.jpeg" alt="Product 3">
          <h3>Product 3</h3>
          <p>$30.00</p>
        </a>
      </li>
    </ul>
  </section>
</main>
```

```
  <footer>
    <p>&copy; 2023 Sammie Smith</p>
  </footer>
</body>
</html>
```

CSS Code:

```
/* Global Styles */
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
  }
    body {
    font-family: Arial, sans-serif;
    font-size: 16px;
    line-height: 1.5;
  }
    a {
    text-decoration: none;
  }
    ul {
    list-style: none;
  }
    /* Header Styles */
  header {
    background-color: #333;
```

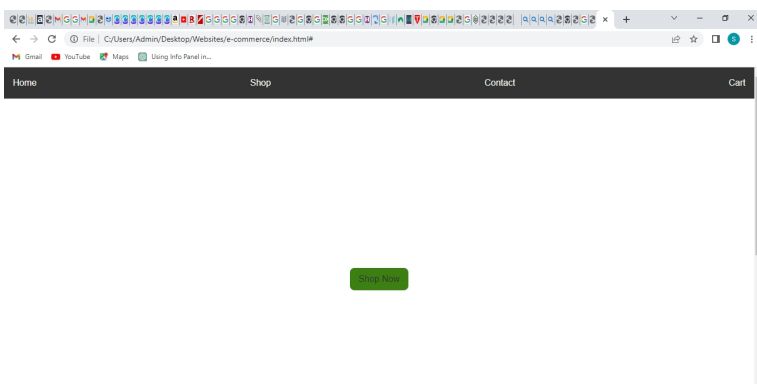
```
  color: #d6d4d4;
  padding: 1rem;
}
  nav ul {
  display: flex;
  justify-content: space-between;
}
  nav ul li {
  margin-right: 1rem;
}
  nav ul li:last-child {
  margin-right: 0;
}
  nav ul li a {
  color: #fff;
}
  nav ul li a:hover {
  text-decoration: underline;
}
  /* Hero Styles */
.hero {
  background-image: url('https://via.placeholder.com/1920x1080');
  background-size: cover;
  background-position: center center;
  height: 500px;
  display: flex;
```

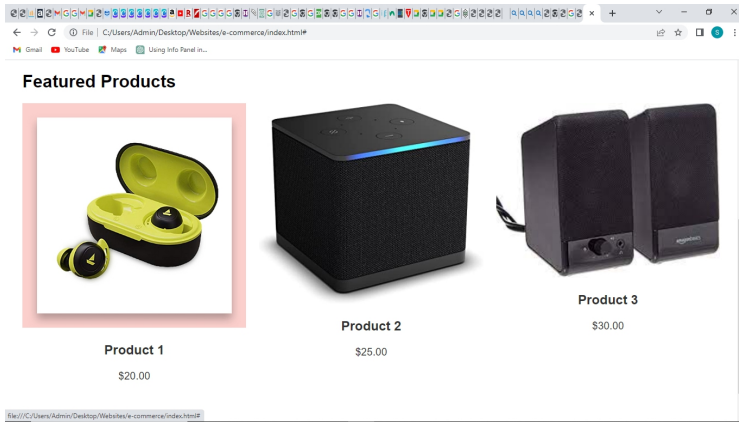
```
  flex-direction: column;
  justify-content: center;
  align-items: center;
  text-align: center;
  color: #fff;
}
  .hero h1 {
  font-size: 3rem;
  margin-bottom: 1rem;
}
  .hero p {
  font-size: 1.5rem;
  margin-bottom: 1.5rem;
}
  .btn {
  background-color: #3c8014;
  color: #333;
  padding: 0.5rem 1rem;
  border-radius: 0.5rem;
  font-size: 1rem;
  transition: background-color 0.3s ease;
}
  .btn:hover {
  background-color: #1f725d;
  color: #f3f3f5;
  cursor: pointer;
```

```
}
  /* Featured Products Styles */
.featured-products {
  padding: 2rem;
}
  .featured-products h2 {
  font-size: 2rem;
  margin-bottom: 1rem;
}
  .featured-products ul {
  display: flex;
  justify-content: space-between;
  flex-wrap: wrap;
}
  .featured-products li {
  margin-bottom: 2rem;
  flex-basis: calc(33.33% - 1rem);
}
  .featured-products li a {
  display: block;
  text-align: center;
  color: #333;
}
  .featured-products li h3 {
  margin: 1rem 0;
  font-size: 1.5rem;
```

```
}
  .featured-products li p {
  margin-bottom: 1rem;
  font-size: 1.2rem;
}
  .featured-products li img {
  width: 100%;
  height: auto;
}
  /* Footer Styles */
footer {
  background-color: #adacac;
  padding: 1rem;
  text-align: center;
}
```

See the results of the web view below.





CHAPTER TWELVE

GETTING STARTED WITH JAVASCRIPT

WHAT EXACTLY IS JAVASCRIPT

JavaScript is a high-level, interpreted programming language that is widely used for client-side web development. It was first developed by Brendan Eich at Netscape in 1995 as a way to add dynamic and interactive features to web pages.

When designing web pages, applications, servers, or even games, web developers frequently employ the lightweight programming language JavaScript to generate more dynamic interactions.

JavaScript is typically used by developers along with HTML and CSS. For formatting HTML components, the scripting language and CSS function well together. User interaction is still maintained via JavaScript, but CSS is unable to do so on its own.

The scripting language JavaScript is used for creating websites, mobile apps, and video games. You can achieve this by exploring free JavaScript templates and applications on code hosting platforms like GitHub or using learning platforms like BitDegree. JavaScript is a versatile language that can be used for a variety of purposes, including building web applications, creating browser extensions, and developing mobile apps. It is also used on the server-side, with platforms like Node.js that allow developers to build full-stack JavaScript applications. JavaScript is supported by all modern web browsers, making it an essential tool for web developers. It is used to add interactivity to web pages, including form validation, pop-up windows, and animations. It can also be used to retrieve and display data from web servers, allowing web pages to be updated dynamically without requiring a page refresh. JavaScript is an object-oriented language, which means that it allows developers to define and create their objects with properties and methods. It also supports functional programming, which allows developers to write code in a more modular and reusable way. JavaScript is a versatile and powerful programming language that is essential for web development. It allows developers to create dynamic and interactive web pages and has become an indispensable tool for building modern web applications.

HISTORY OF JAVASCRIPT

Brendan Eich invented JavaScript in 1995 while working at Netscape Communications. For use with Netscape Navigator, the company's primary web browser, Netscape and Eich created JavaScript as a scripting language. LiveScript was the original name of the language, which Netscape changed to JavaScript to present it as a complement to the Java language, a creation of their partner Sun Microsystems. However, JavaScript and the Java programming language are unrelated, save for some superficial grammatical similarities.

As soon as JavaScript was released, more and more browsers began

to support it. Despite this, for a significant portion of its history, JavaScript was not regarded as a serious programming language. Although there were significant speed and security problems in its early editions, there were no other options for developers. They needed to use JavaScript to run programs within the browser.

A significant turning point for JavaScript occurred in 2008 with the release of Google's high-performance, open-source Chrome V8 JavaScript engine. The following spread of quick JavaScript engines allowed programmers to create complex browser-based applications with performance on par with desktop and mobile apps.

Shortly after, Ryan Dahl published the Node.js open-source, cross-platform environment. It offered a method for executing JavaScript code from outside of a browser. It facilitated JavaScript's present prominence by releasing it from the restrictions of the browser. Today, JavaScript may be used to create a wide range of apps, including those for desktops, mobile, servers, and browsers. The majority of the biggest internet businesses today, such as Facebook, Twitter, Netflix, and Google, all employ JavaScript in their offerings.

FEATURES OF JAVASCRIPT

JavaScript has a wide range of features that make it a popular and powerful programming language. Some of its key features include:

- **Client-side scripting**: JavaScript is primarily used as a client-side scripting language, which means it runs on the user's web browser and allows for dynamic and interactive web page content.
- **Cross-platform compatibility**: JavaScript is supported by all major web browsers, as well as mobile devices and desktop applications, making it a widely accessible language.
- **Object-oriented programming**: JavaScript is an object-oriented language, that allows developers to define and create their objects with properties and methods.
- **Functional programming**: JavaScript supports functional programming, which allows developers to write code in a more modular and reusable way.
- **Asynchronous programming**: JavaScript supports

asynchronous programming, which allows for non-blocking code execution and can improve performance in certain scenarios.

- **Event-driven programming**: JavaScript is event-driven, which means it can respond to user actions or other events on a web page.
- **Dynamic typing**: JavaScript is a dynamically typed language, which means that variable types are determined at runtime rather than explicitly defined.
- **Easy integration with HTML and CSS**: JavaScript can be easily integrated with HTML and CSS to create dynamic and interactive web pages.
- **Extensive libraries and frameworks**: JavaScript has a large and growing ecosystem of libraries and frameworks, making it easy to build complex applications and websites.
- **Interpreter-Based:** In contrast to compiled languages, JavaScript is an interpreted language. It is comparable to Python and Ruby in that regard. Line by line, the source code of JavaScript is interpreted and executed by the browser. A compiled language, on the other hand, needs to be translated into an executable byte-code. Examples of compiled languages are Java and C++.
- **Case Sensitive:** JavaScript pays close attention to case. Each word in all keywords, variables, function names, and other identifiers must be consistently capitalized. These variables Due to the distinction in the cases, hitCounter, and hitcounter are both different variables. Additionally, all keywords, including "var," are case-sensitive.
- **Control Statements:** Control statements like if-else-if, switch-case, and loops like for, while, and do-while loops are available in JavaScript. Its user can create complex logic using this programming language's powerful control statements.
- **Client-side validations:** JavaScript is heavily used for easing the interactions between users and web applications. Towards this, validation plays a significant role. Validations can guide users to fill forms correctly with valid data and prevent spam submissions.

- **Prototype-based:** JavaScript employs the prototypal inheritance model rather than the more well-known class inheritance because it is a prototype-based scripting language. As a result, we define the Objects prototype and utilize this object prototype to build other Objects of the same type rather than defining classes and deriving objects from them. Given that this behavior resembles that of the Object factory function, this design pattern is frequently used in JavaScript applications.
- **Object Shorthand:** Following ES6/ES2015 syntax, you can use the shorthand and just give the key name to define an object whose keys have the same name as the variables passed in as properties. By enabling you to use this shorthand, the object shorthand allows you to save time and space while also improving the idiomaticity and conciseness of the code.

DIFFERENCES BETWEEN JAVASCRIPT AND OTHER PROGRAMMING LANGUAGES

JavaScript is a unique programming language with some key differences compared to other languages. Here are a few of the main differences between JavaScript and other programming languages:

- **Interpretation**: JavaScript is an interpreted language, which means that the code is executed directly by the browser or runtime environment. This is in contrast to compiled languages like C++, which require compilation before they can be executed.
- **Dynamic typing**: JavaScript is a dynamically typed language, meaning that the variable types are determined at runtime rather than explicitly defined. This is in contrast to statically typed languages like Java, where variable types are defined at compile-time.
- **Prototypal inheritance**: JavaScript uses prototypal inheritance, which is different from the classical inheritance used in languages like Java and C++. Prototypal inheritance allows objects to inherit

properties and methods directly from other objects.

- **Event-driven programming**: JavaScript is an event-driven language, meaning that it can respond to user actions or other events on a web page. This is in contrast to procedural languages like C, which follow a linear sequence of code execution.
- **Client-side scripting**: JavaScript is primarily used as a client-side scripting language, running in the user's web browser rather than on a server. This is in contrast to server-side languages like PHP or Ruby, which run on a server to generate web pages dynamically.
- **Functional programming**: JavaScript supports functional programming, which is a different programming paradigm than the object-oriented programming used in many other languages.
- **Versatility**: JavaScript is a versatile language that can be used for a wide range of applications beyond web development, including server-side programming with Node.js and desktop or mobile app development with frameworks like Electron or React Native.

UNDERSTANDING OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a programming paradigm that involves organizing code into objects that have attributes and behaviors. It's based on the idea of creating real-world models of objects in code. OOP focuses on the idea of creating reusable code and simplifying complex systems.

In OOP, objects have their properties and methods, and they can interact with other objects to perform specific tasks. OOP has four fundamental concepts: encapsulation, inheritance, abstraction, and polymorphism.

- Encapsulation is the concept of hiding the implementation details of an object and only exposing the necessary methods and properties. This helps to ensure that the object can only be used in ways that were intended by the designer.
- Inheritance is the concept of creating new classes based

on existing ones. Inheritance allows a subclass to inherit the properties and methods of a superclass. This allows developers to create more specialized classes without having to rewrite code that already exists.

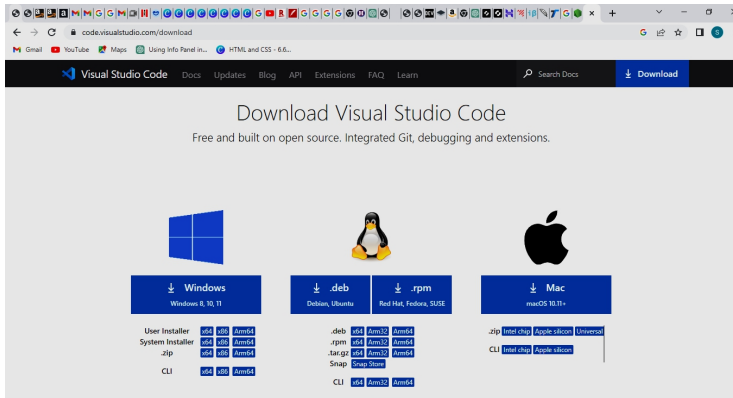
- Abstraction is the concept of simplifying complex systems by breaking them down into smaller, more manageable parts. Abstraction allows developers to focus on the essential features of an object, without worrying about the implementation details.
- Polymorphism is the concept of creating code that can work with objects of different types. This allows developers to write more flexible and reusable code.

OOP is used in many popular programming languages, including Java, Python, and C++. It has become a standard for creating robust and scalable software applications. Understanding OOP is essential for any developer who wants to build complex systems that are easy to maintain and extend.

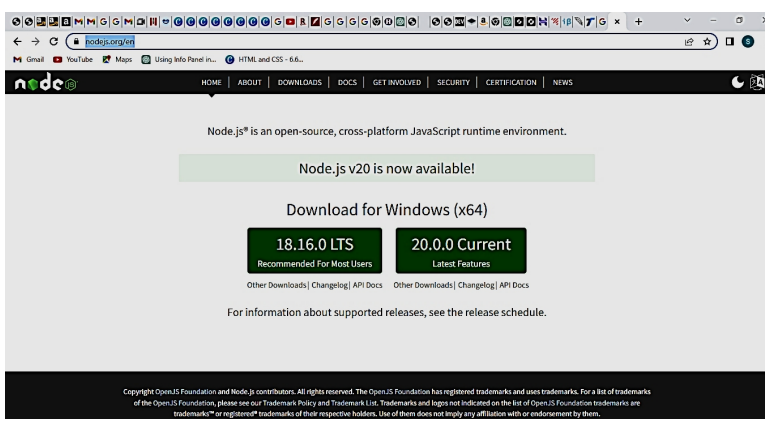
SETTING UP JAVASCRIPT DEVELOPMENT ENVIRONMENT

To set up a JavaScript development environment, you will need a few key tools and software. Here are the basic steps to get started:

- **Install a code editor**: A code editor is a tool that allows you to write and edit code. There are many code editors available, including free and paid options. Some popular choices for JavaScript development include Visual Studio Code, Sublime Text, and Atom. You can download and install Visual Studio at “https://code.visualstudio.com/download”.

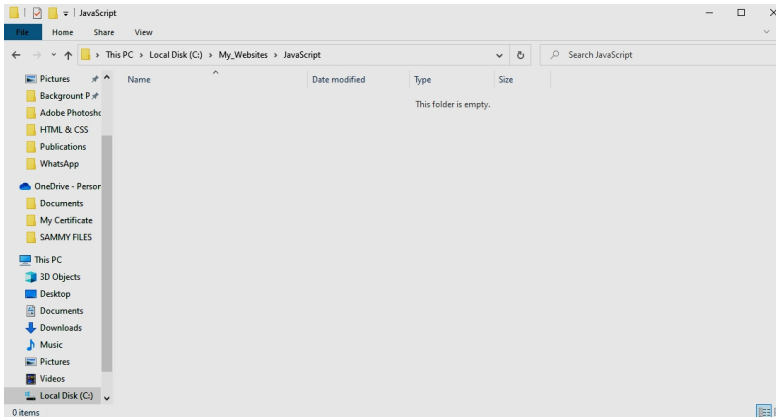


- **Install Node.js**: Node.js is a JavaScript runtime environment that allows you to run JavaScript code outside of a web browser. It includes a package manager called npm, which is used to manage and install third-party packages and libraries. You can download Node.js from the official website “https://nodejs.org/en” and follow the installation instructions for your operating system. Install the "Node.js" runtime environment on your system. Make sure to install the LTS version for stability and long-term support.



- **Create a project directory**: Once you have a code editor and Node.js installed, you can create a project directory where you will store your JavaScript code files. You can create a new directory using the command line or through your operating system's file

manager.



- **Initialize a new Node.js project**: In your project directory, you can initialize a new Node.js project using the npm init command in the terminal. This will create a package.json file that includes information about your project and its dependencies.
- **Install dependencies**: You can use npm to install any dependencies or packages that you need for your project. For example, if you are building a web application, you might want to install a package like Express.js to handle server-side routing and middleware.
- **Start coding**: With your development environment set up, you can start writing JavaScript code. You can create a new JavaScript file in your project directory and begin writing code using your code editor.

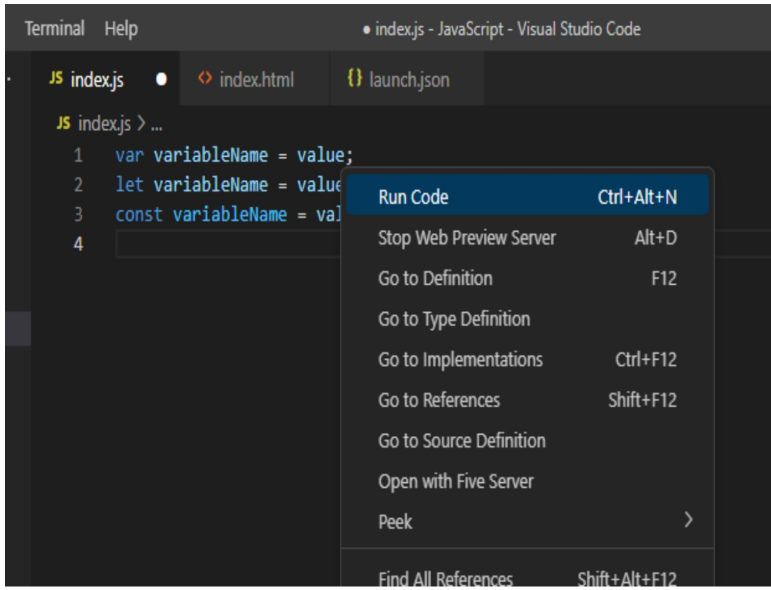
You can write JavaScript code using a simple editor like Notepad. However, you can install any open-sourced or licensed IDE (Integrated Development Environment) to get the advantage of IntelliSense support for JavaScript and syntax error/warning highlighter for rapid development.

HOW TO CONFIGURE VISUAL STUDIO CODE TO RUN JAVASCRIPT

To configure Visual Studio Code to run JavaScript code, you can

follow these steps:

- Launch Visual Studio Code.
- Install the "Code Runner" extension in Visual Studio Code. You can do this by clicking on the Extensions icon in the left sidebar, searching for "Code Runner," and clicking on the "Install" button.
- Open the JavaScript file that you want to run in Visual Studio Code.
- Press the "Ctrl + Shift + P" key combination to open the command palette. Search for the "Code Runner: Run Code" command and select it.
- Alternatively, you can use the "Ctrl + Alt + N" key combination to run the code directly.
- If the JavaScript file requires input from the user, you can enter the input in the terminal that opens up after running the code.
- You can also configure the Code Runner extension settings by going to File > Preferences > Settings, searching for "code-runner," and modifying the options to suit your needs.
- When you carefully completed the steps above you can now run the JavaScript file on your Visual Studio Code by right-clicking the code area, then choose, Run Code. See the screenshot below.



CHAPTER THIRTEEN

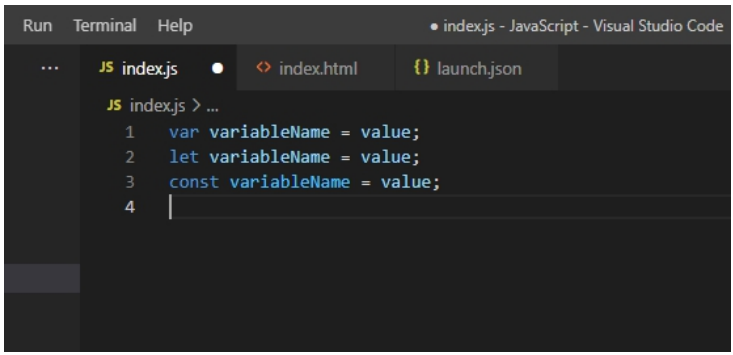
JAVASCRIPT BASICS

Variables and data types are fundamental concepts in programming. In JavaScript, variables are used to store data values that can be used later in the program. Data types, on the other hand, are classifications that define the types of values that can be stored in a variable. In this chapter, we will explore these concepts in depth, with practical examples.

JAVASCRIPT VARIABLES

In JavaScript, variables are declared using the "var", "let", or "const" keywords. These keywords tell the JavaScript interpreter that a new

variable is being declared. The syntax for declaring a variable is as follows:



Here, "variableName" is the name of the variable, and "value" is the initial value that is assigned to the variable. The "var" keyword is used to declare a variable that has a function-level scope, while the "let" keyword is used to declare a variable that has a block-level scope. The "const" keyword is used to declare a constant variable, which means its value cannot be changed once it is assigned.

JAVASCRIPT DATATYPES

JavaScript has several data types, including strings, numbers, booleans, objects, and arrays.

Strings:

Strings are a data type used to represent text. They are enclosed in quotes, either single or double. Here's an example:

var myString = "Hello, world!";

Numbers:

Numbers are used to represent numerical values. They can be integers or floating-point numbers. Below is an example:

var myNumber = 42;

Booleans:

Booleans are used to represent true or false values. They are typically used in conditional statements. Below is an example:

var myBoolean = true;

Objects:

Objects are complex datatypes that can contain multiple properties and methods. They are created using the "object" keyword, and properties are defined using the syntax "propertyName: value". Here's an example:

var myObject = { name: "John", age: 30, occupation: "Programmer" };

Arrays:

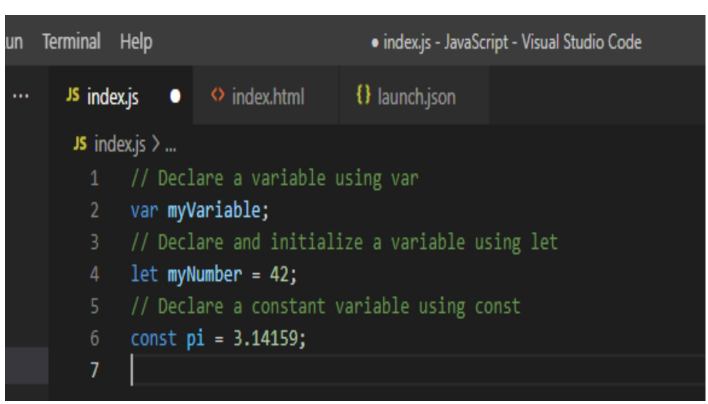
Arrays are used to store a collection of values, which can be of any data type. They are created using the "array" keyword, and each value is separated by a comma. Here's an example:

var myArray = [1, 2, 3, "four", true];

DECLARING VARIABLES

To declare a variable in JavaScript, use var, let, or const keywords followed by the name of the variable you want to create. You can also initialize the variable by assigning a value to it, as shown in the examples above. If you don't assign a value to the variable when you declare it, its value will be undefined.

Let's take a look at some examples of how to declare variables in JavaScript:


un Terminal Help
• index.js - JavaScript - Visual Studio Code
index.js
index.html
launch.json
index.js > ...
1 // Declare a variable using var
2 var myVariable;
3 // Declare and initialize a variable using let
4 let myNumber = 42;
5 // Declare a constant variable using const
6 const pi = 3.14159;
7

WORKING WITH STRINGS, NUMBERS, AND

BOOLEAN VALUES

Strings, numbers, and boolean values are the most common data types used in JavaScript. Understanding how to work with these data types is essential to effectively write JavaScript code.

WORKING WITH STRINGS

A string is a sequence of characters enclosed in single or double quotes. In JavaScript, strings are considered objects and have a set of properties and methods that can be used to manipulate them.

Declaring a String Variable

To declare a string variable in JavaScript, you can use either single or double quotes. Here is an example:

let message = 'Hello World!';

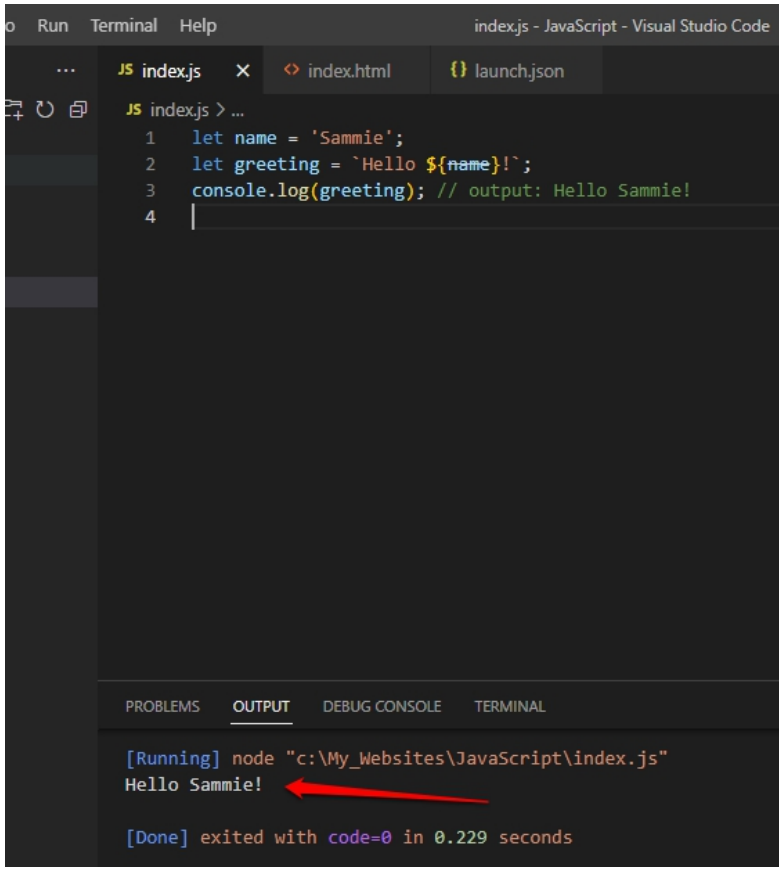
You can also use backticks to declare a string variable. Backticks are useful for creating template literals, which allow you to embed expressions inside a string. Here is an example:

let name = 'Sammie';

let greeting = `Hello ${name}!`;

console.log(greeting); // output: Hello Sammie!

See the output below.



Accessing Individual Characters in a String

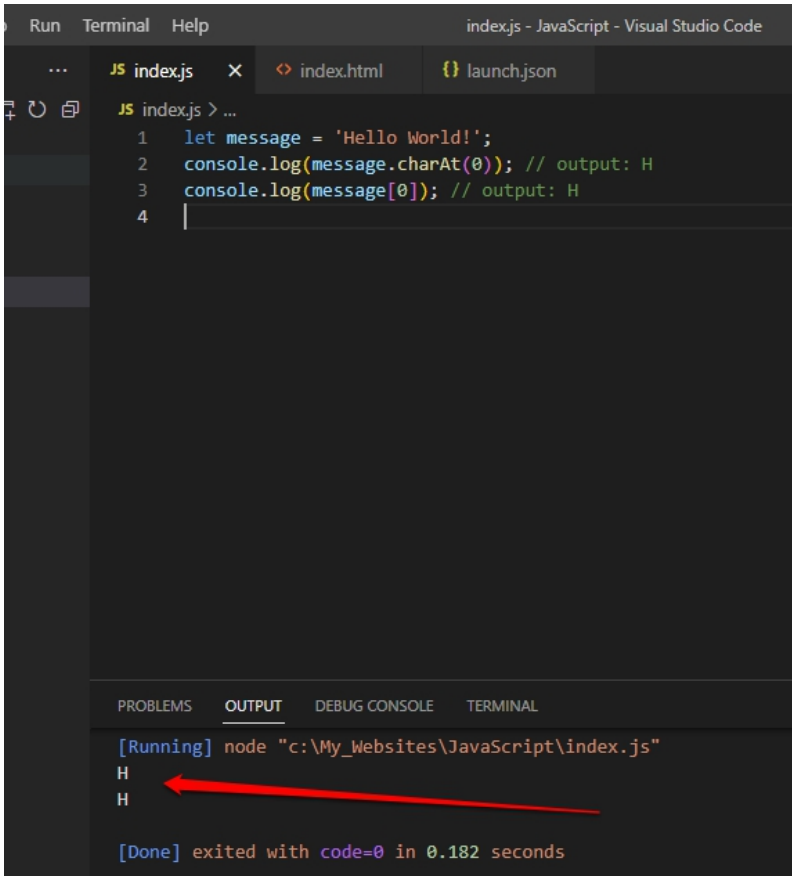
You can access individual characters in a string by using the charAt() method or square bracket notation. Here is an example:

let message = 'Hello World!';

console.log(message.charAt(0)); // output: H

console.log(message[0]); // output: H

See the output below.

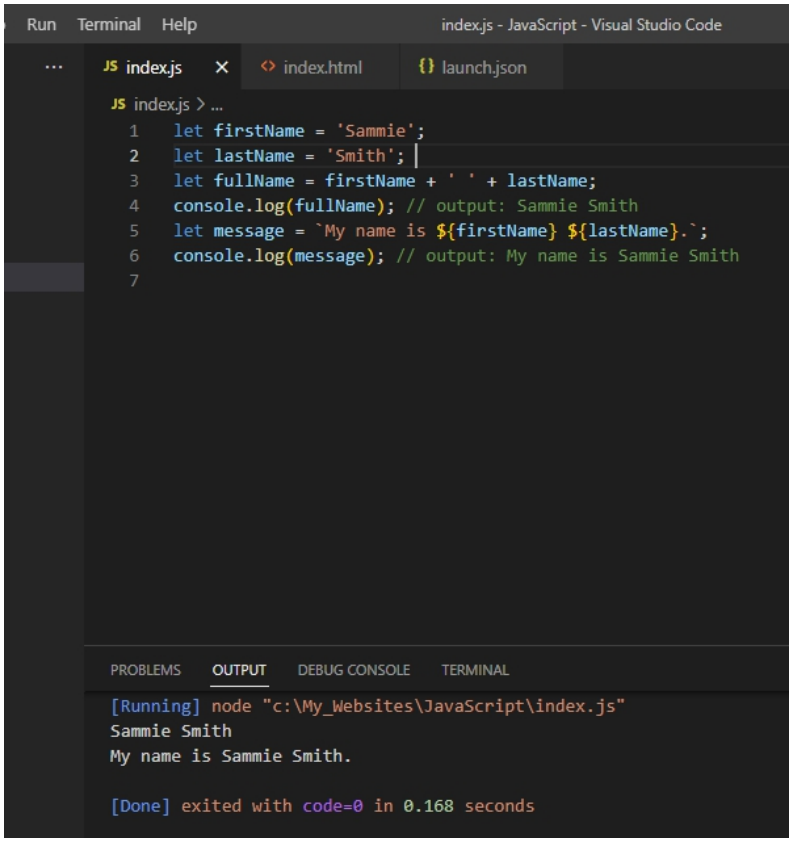


Concatenating Strings

You can concatenate two or more strings using the concatenation operator (+) or template literals. Below is an example:

```
let firstName = 'Sammie';
let lastName = 'Smith';
let fullName = firstName + ' ' + lastName;
console.log(fullName); // output: Sammie Smith
let message = `My name is ${firstName} ${lastName}.`;
console.log(message); // output: My name is Sammie Smith
```

See the output below.


Run Terminal Help
index.js - JavaScript - Visual Studio Code
index.js
index.html
launch.json
index.js > ...
1 let firstName = 'Sammie';
2 let lastName = 'Smith';
3 let fullName = firstName + ' ' + lastName;
4 console.log(fullName); // output: Sammie Smith
5 let message = `My name is ${firstName} ${lastName}.`;
6 console.log(message); // output: My name is Sammie Smith
7
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
[Running] node "c:\My_Websites\JavaScript\index.js"
Sammie Smith
My name is Sammie Smith.
[Done] exited with code=0 in 0.168 seconds

WORKING WITH NUMBERS

In JavaScript, numbers are represented as floating-point values. You can perform arithmetic operations such as addition, subtraction, multiplication, and division of numbers.

Declaring a Number Variable

To declare a number variable in JavaScript, you can simply assign a numeric value to a variable. Here is an example:

let x = 10; let y = 3.14;

Performing arithmetic operations: You can perform arithmetic operations on numbers using operators such as +, -, *, and /. Here is an example:

let x = 10; let y = 5;

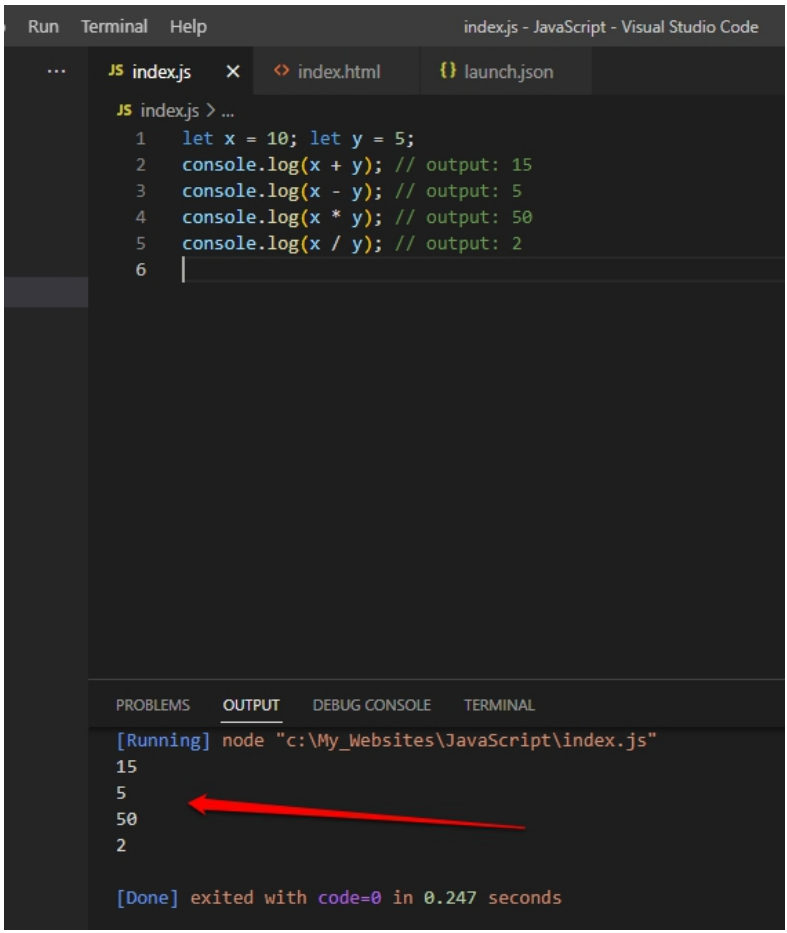
console.log(x + y); // output: 15

console.log(x - y); // output: 5

*console.log(x * y); // output: 50*

console.log(x / y); // output: 2

See the output below.



WORKING WITH BOOLEANS

A Boolean value represents either true or false. Booleans are commonly used in conditional statements and loops to control the flow of a program.

Declaring a Boolean variable: To declare a Boolean variable in JavaScript, you can simply assign a true or false value to a variable. Here is an example:

let isRaining = true;

let isSunny = false;

Using Boolean Operators

You can use Boolean operators such as && (and), || (or), and ! (not) to perform logical operations on Boolean values. Here is an example:

let isRaining = true;

let isCold = true;

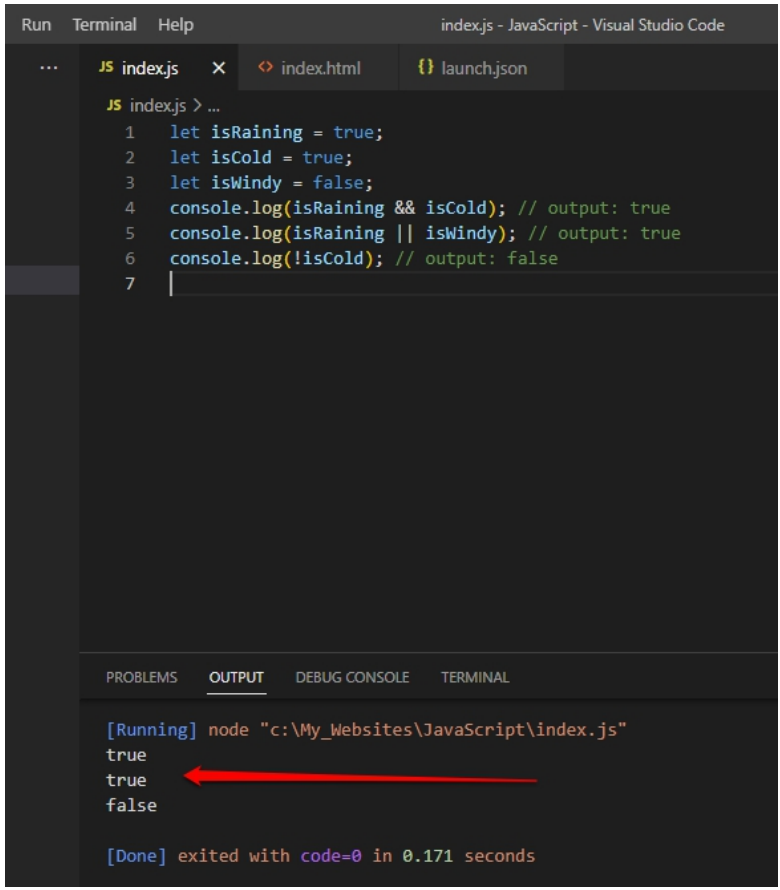
let isWindy = false;

console.log(isRaining && isCold); // output: true

console.log(isRaining || isWindy); // output: true

console.log(!isCold); // output: false

See the output below.



UNDERSTANDING OBJECTS AND ARRAYS

In JavaScript, objects and arrays are two important data structures that allow developers to store and manipulate data in various ways. In this section, we will explore the basics of objects and arrays, their syntax, and how to use them in practical examples.

Understanding Objects

An object in JavaScript is a collection of properties, which are key-value pairs that describe various characteristics of the object. For example, a person's object can have properties like name, age, and gender, each with its value. Objects can contain any type of data, including other objects, functions, and arrays.

Declaring Objects

To declare an object in JavaScript, we use the curly braces ({}) notation. Inside the curly braces, we list the properties of the object, each separated by a comma. The key of the property is separated from the value using a colon (:).

Here is an example of an object that represents a person:

let person = { name: "John Doe", age: 30, gender: "Male" };

In the example above, we declared a person object with three properties: name, age, and gender. The value of the name property is "John Doe", age is 30, and gender is "Male". We can access the properties of an object using the dot notation, like this:

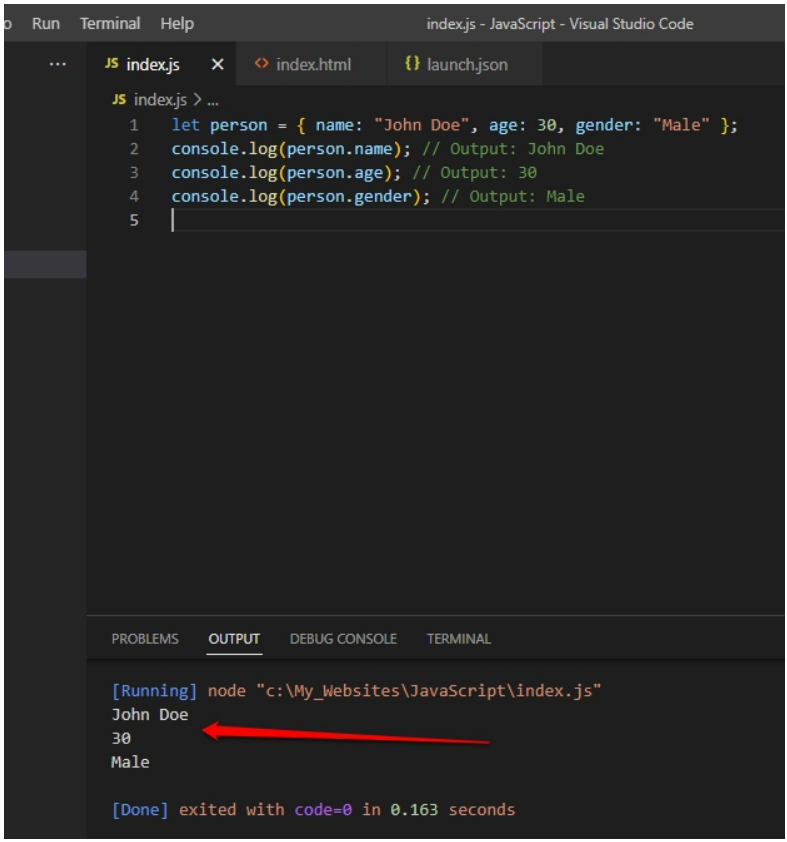
let person = { name: "John Doe", age: 30, gender: "Male" };

console.log(person.name); // Output: John Doe

console.log(person.age); // Output: 30

console.log(person.gender); // Output: Male

See the output below.



We can also add new properties to an object or update existing ones, like this:

let person = { name: "John Doe", age: 30, gender: "Male" };

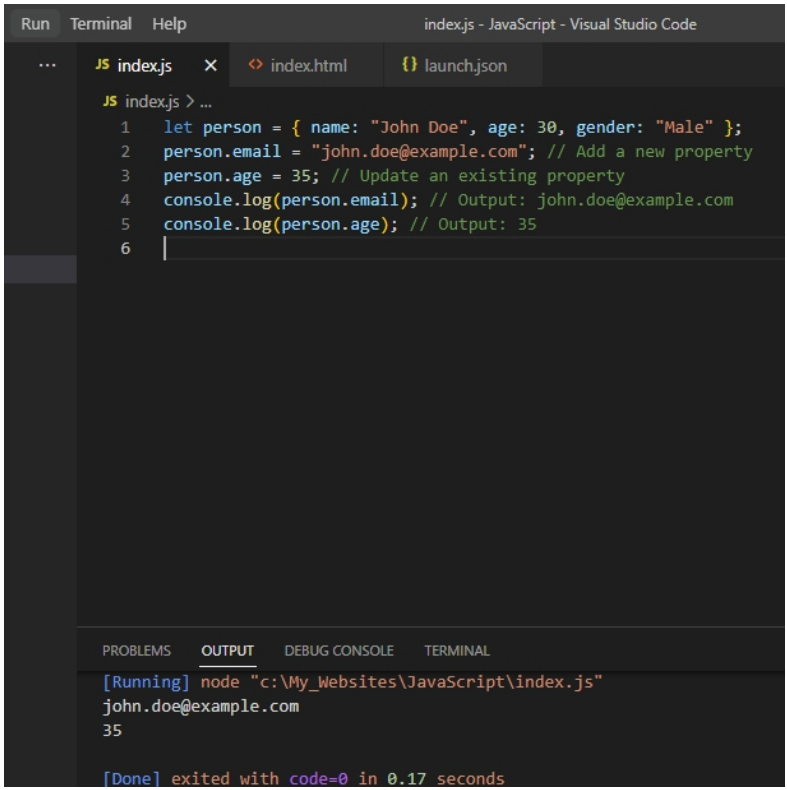
person.email = "john.doe@example.com"; // Add a new property

person.age = 35; // Update an existing property

console.log(person.email); // Output: john.doe@example.com

console.log(person.age); // Output: 35

See the output below.


Run Terminal Help
index.js - JavaScript - Visual Studio Code
index.js
index.html
launch.json
index.js > ...
1 let person = { name: "John Doe", age: 30, gender: "Male" };
2 person.email = "john.doe@example.com"; // Add a new property
3 person.age = 35; // Update an existing property
4 console.log(person.email); // Output: john.doe@example.com
5 console.log(person.age); // Output: 35
6
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[Running] node "c:\My_Websites\JavaScript\index.js"
john.doe@example.com
35
[Done] exited with code=0 in 0.17 seconds

Understanding Arrays

An array in JavaScript is a collection of values, which can be of any data type, including other arrays and objects. Arrays are ordered, meaning that the order in which the values are stored is important.

Declaring Arrays

To declare an array in JavaScript, we use the square brackets ([]) notation. Inside the square brackets, we list the values of the array, each separated by a comma.

Here is an example of an array that contains three numbers:

let numbers = [1, 2, 3];

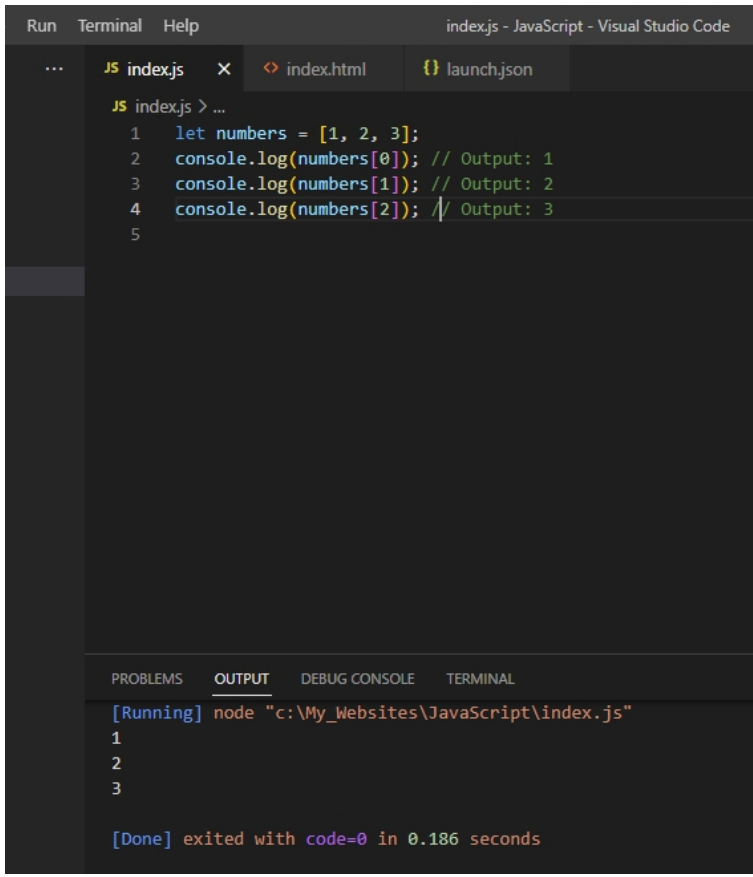
We can access the values of an array using their index, which starts at 0. For example:

let numbers = [1, 2, 3];

console.log(numbers[0]); // Output: 1

console.log(numbers[1]); // Output: 2

console.log(numbers[2]); // Output: 3



We can also add new values to an array or update existing ones, like this:

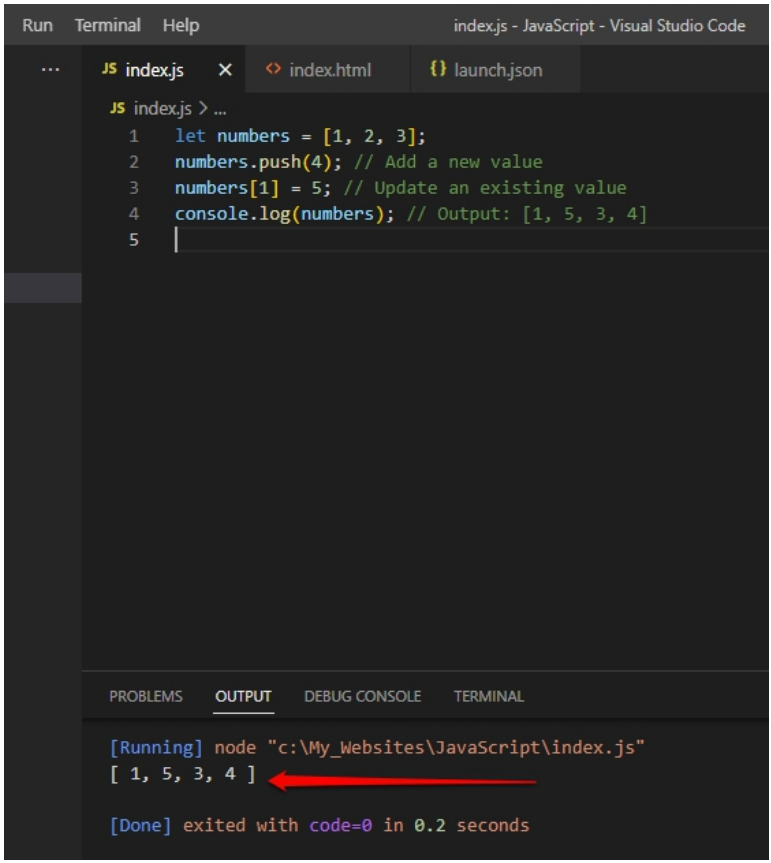
let numbers = [1, 2, 3];

numbers.push(4); // Add a new value

numbers[1] = 5; // Update an existing value

console.log(numbers); // Output: [1, 5, 3, 4]

See the output below.



Arrays can also contain objects or other arrays, like this:

let people = [{ name: "John Doe", age: 30 }, { name: "Jane Doe", age: 25 }];

console.log(people[0].name); // Output: John Doe

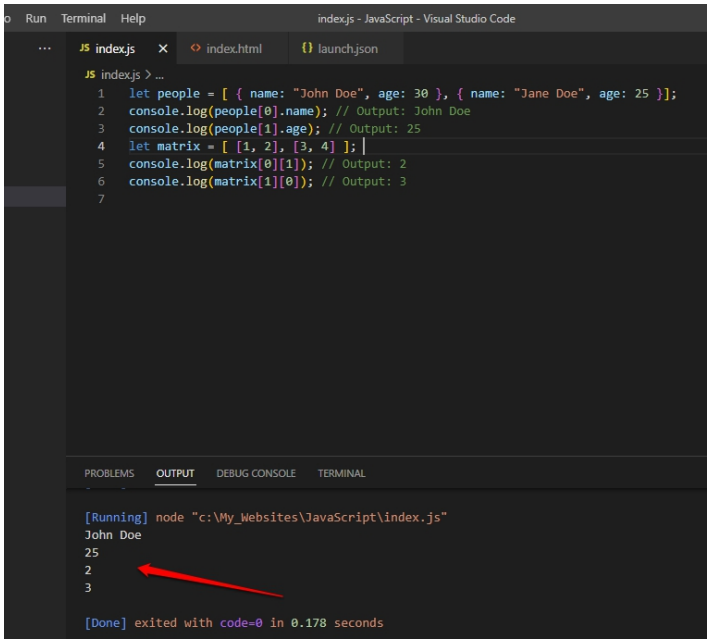
console.log(people[1].age); // Output: 25

let matrix = [[1, 2], [3, 4]];

console.log(matrix[0][1]); // Output: 2

console.log(matrix[1][0]); // Output: 3

See the output below.



CHAPTER FOURTEEN

OPERATORS AND EXPRESSIONS

In this chapter, we will discuss operators and expressions in JavaScript. Operators are symbols or keywords that are used to perform operations on values or variables. Expressions, on the other hand, are a combination of operators, values, and variables that produce a result.

ARITHMETIC OPERATORS

JavaScript has several arithmetic operators that can be used to perform mathematical operations. These operators include addition (+), subtraction (-), multiplication (*), division (/), and modulo (%). Let's take a closer look at each of these operators.

Addition (+):

This operator is used to add two or more numbers. For example:

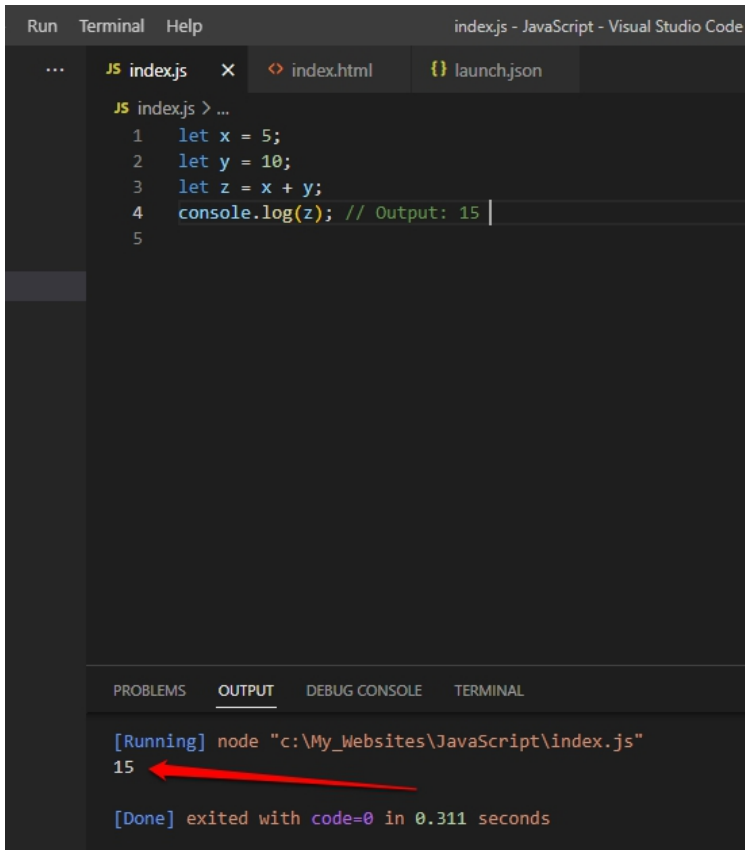
let x = 5;

let y = 10;

let z = x + y;

console.log(z); // Output: 15

See the output below.



Subtraction (-):

This operator is used to subtract one number from another. For example:

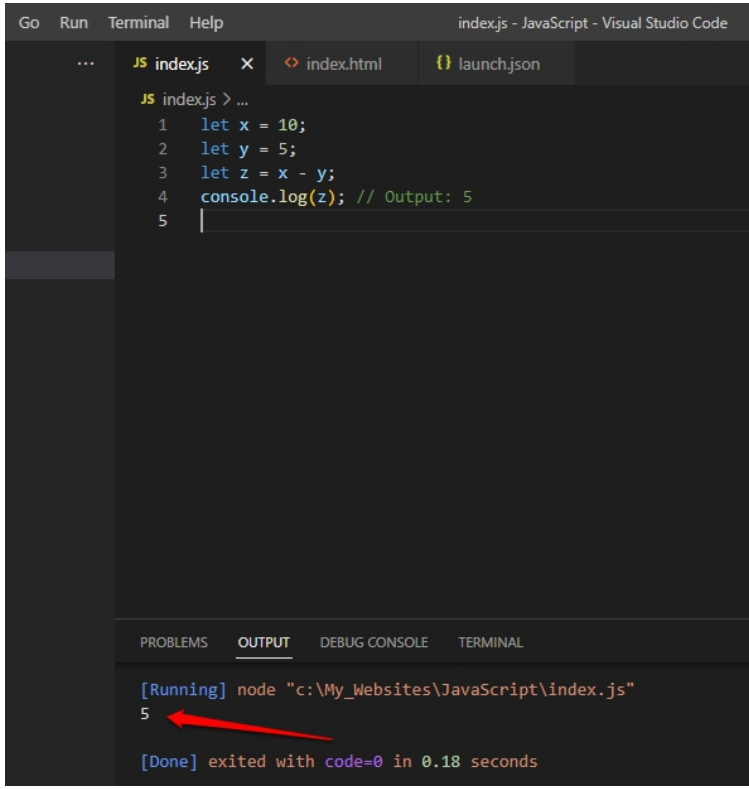
let x = 10;

let y = 5;

let z = x - y;

console.log(z); // Output: 5

See the output below.



Multiplication (*):

This operator is used to multiply two or more numbers. For example:

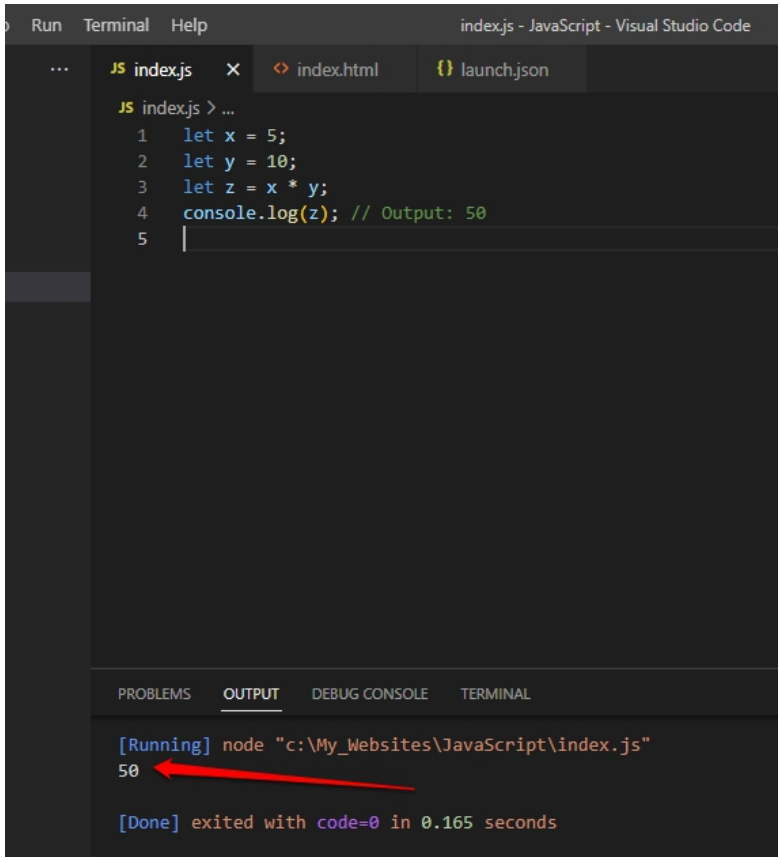
let x = 5;

let y = 10;

*let z = x * y;*

console.log(z); // Output: 50

See the output below.



Division (/):

This operator is used to divide one number by another. For example:

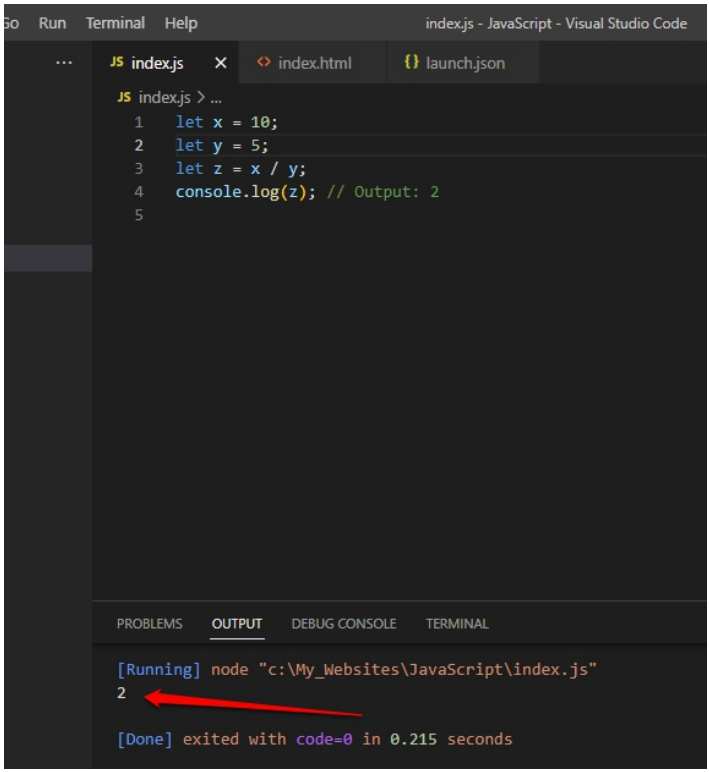
let x = 10;

let y = 5;

let z = x / y;

console.log(z); // Output: 2

See the output below.



Modulo (%):

This operator is used to find the remainder after division. For example:

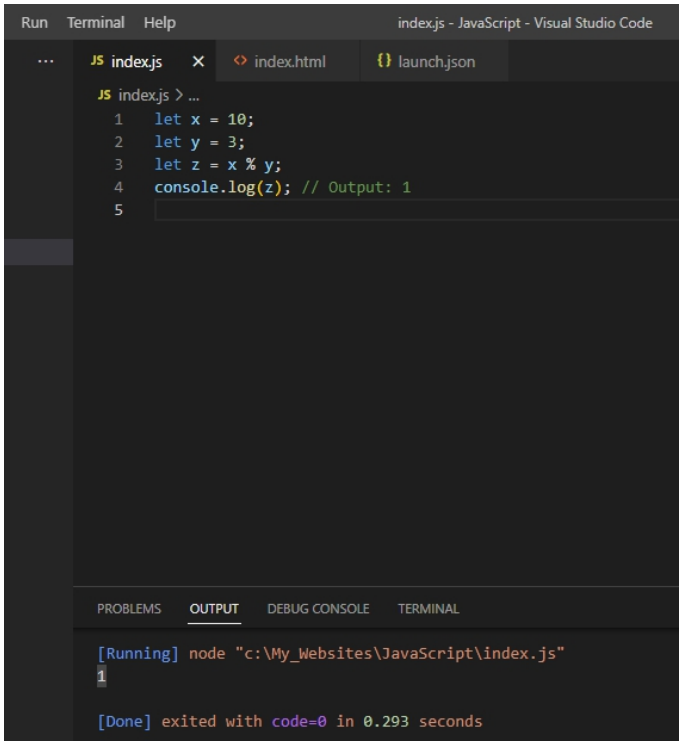
let x = 10;

let y = 3;

let z = x % y;

console.log(z); // Output: 1

See the output below.



COMPARISON OPERATORS

JavaScript also has several comparison operators that can be used to compare two values. These operators include equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). Let's take a closer look at each of these operators.

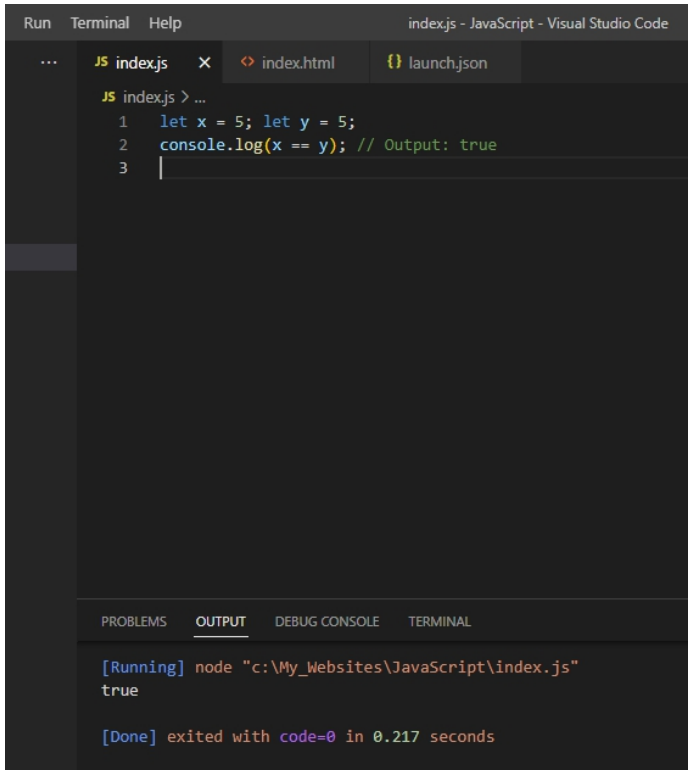
Equal to (==):

This operator is used to check if two values are equal. For example:

let x = 5; let y = 5;

console.log(x == y); // Output: true

See the output below.



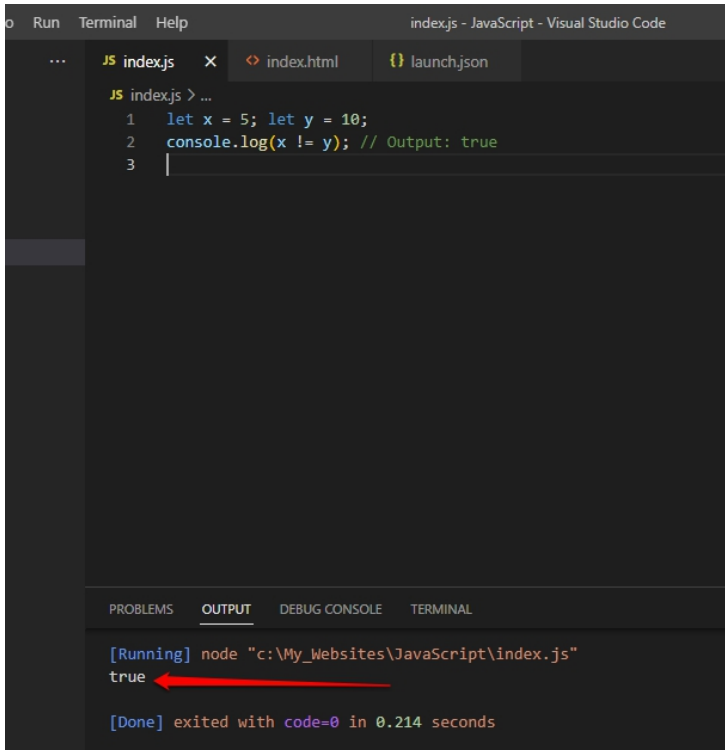
Not Equal to (!=):

This operator is used to check if two values are not equal. For example:

let x = 5; let y = 10;

console.log(x != y); // Output: true

See the output below.



Greater than (>), than (<):

Less These operators are used to compare two values to see if one is greater than or less than the other. For example:

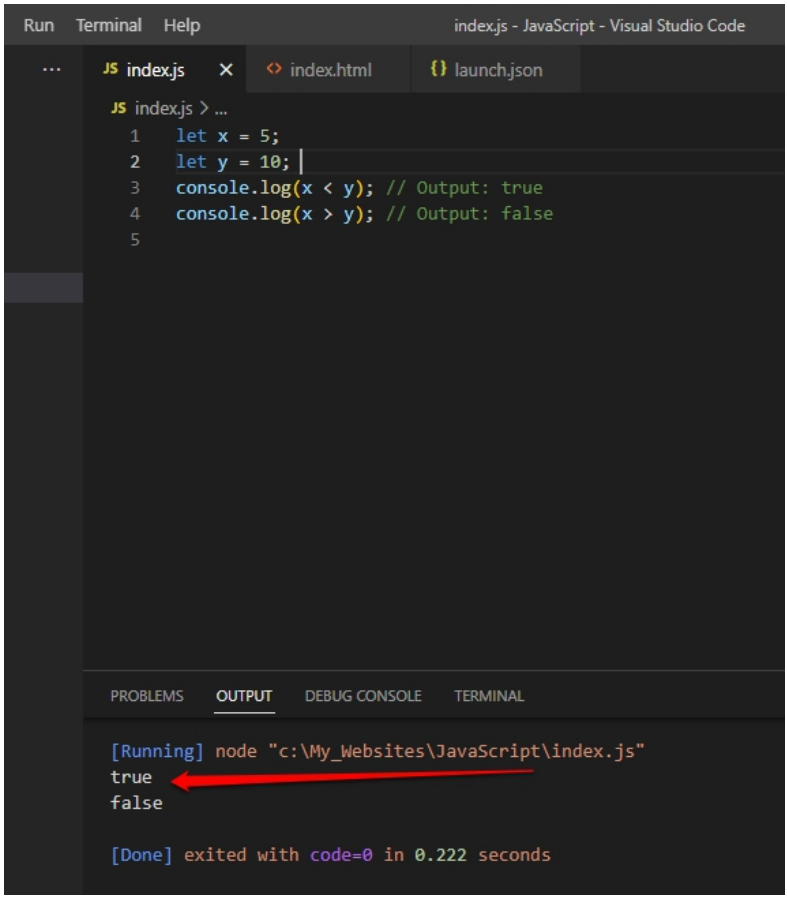
let x = 5;

let y = 10;

console.log(x < y); // Output: true

console.log(x > y); // Output: false

See the output below.



Greater than or equal to (>=), less than or equal to (<=):

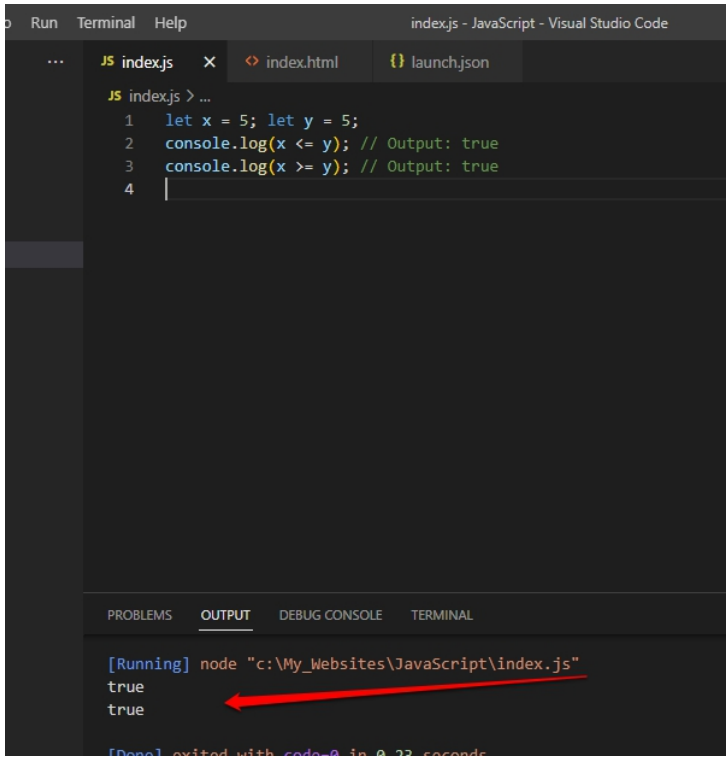
These operators are used to check if one value is greater than or equal to, or less than or equal to, another value. For example:

let x = 5; let y = 5;

console.log(x <= y); // Output: true

console.log(x >= y); // Output: true

see the output below.



LOGICAL OPERATORS

In programming, logical operators are used to combine two or more conditions to form a more complex condition. There are three main logical operators in JavaScript:

AND Operator (&&):

The AND operator is represented by && and is used to check if two conditions are both true. It returns true only if both conditions are true. If one or both conditions are false, it returns false.

Example:

let x = 5;

let y = 10;

if(x > 0 && y > 5) {

console.log("Both conditions are true");

}

else {

console.log("At least one condition is false");

}

// Output: Both conditions are true

In this example, the first condition (x > 0) is true, and the second condition (y > 5) is also true. Therefore, the console will output "Both conditions are true".

OR Operator (||):

The OR operator is represented by || and is used to check if at least one of two conditions is true. It returns true if one or both conditions are true. If both conditions are false, it returns false.

Example:

let x = 5;

let y = 10;

if(x > 0 || y < 5) {

console.log("At least one condition is true");

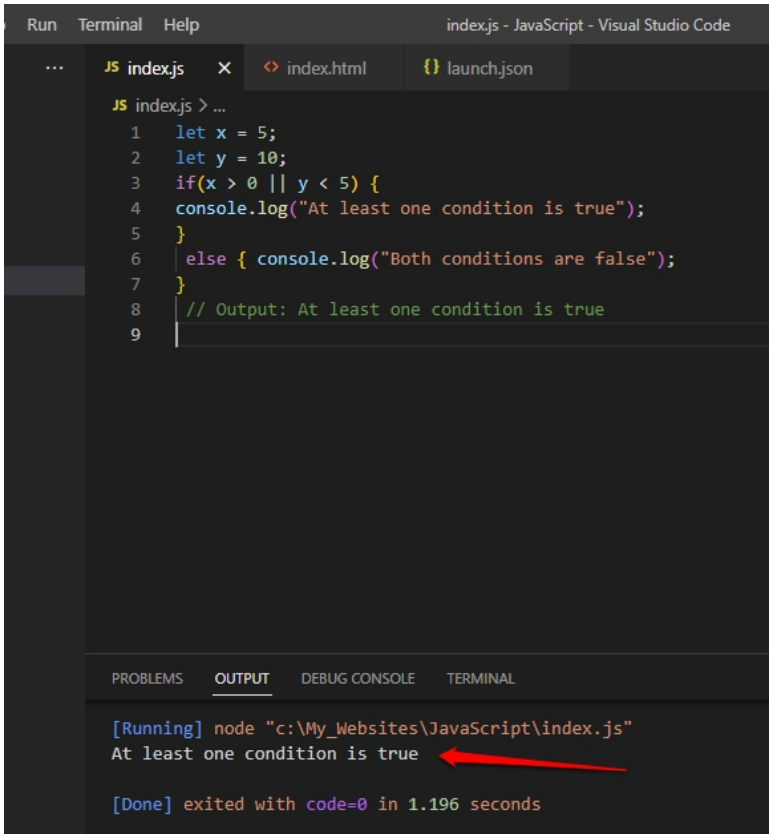
}

else { console.log("Both conditions are false");

}

// Output: At least one condition is true

In this example, the first condition (x > 0) is true, and the second condition (y < 5) is false. However, since at least one of the conditions is true, the console will output "At least one condition is true". See the output below.



NOT Operator (!):

The NOT operator is represented by ! and is used to reverse the logical state of a condition. If a condition is true, the NOT operator will make it false. If a condition is false, the NOT operator will make it true.

Example:

let x = 5;

if(!(x > 0)) {

console.log("The condition is false");

}

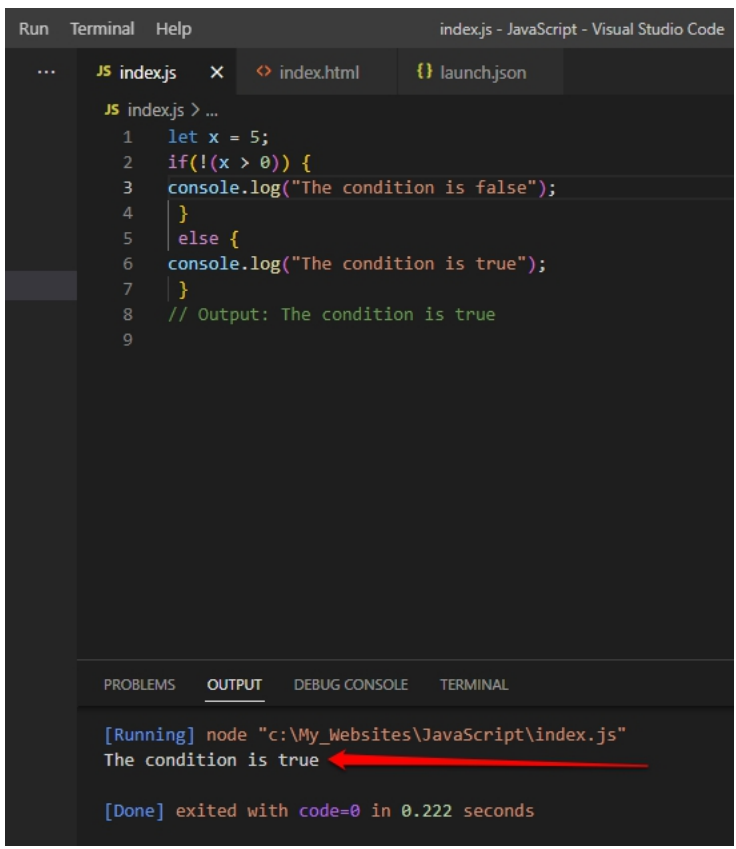
else {

console.log("The condition is true");

}

// Output: The condition is true

In this example, the condition inside the parentheses (x > 0) is true. However, the NOT operator (!) in front of the parentheses reverses the logical state of the condition and makes it false. Therefore, the console will output "The condition is true". See the screenshot below



CONDITIONAL STATEMENTS (IF/ELSE)

Conditional statements, also known as control statements, are used to control the flow of a program. They allow the program to make decisions based on certain conditions. Conditional statements are a crucial part of programming because they make the program more

flexible and intelligent. In this section, we will explore conditional statements in JavaScript.

The two main types of conditional statements are the if statement and the switch statement. The if statement is used to execute a block of code if a condition is true. The switch statement is used to execute different code blocks based on different conditions.

The if Statement:

The if statement syntax in JavaScript is written like this:

if (condition) {

// block of code to be executed if the condition is true

}

The condition in the if statement is an expression that is evaluated to be either true or false. If the condition is true, the block of code inside the curly braces will be executed. If the condition is false, the block of code will be skipped.

Here is an example:

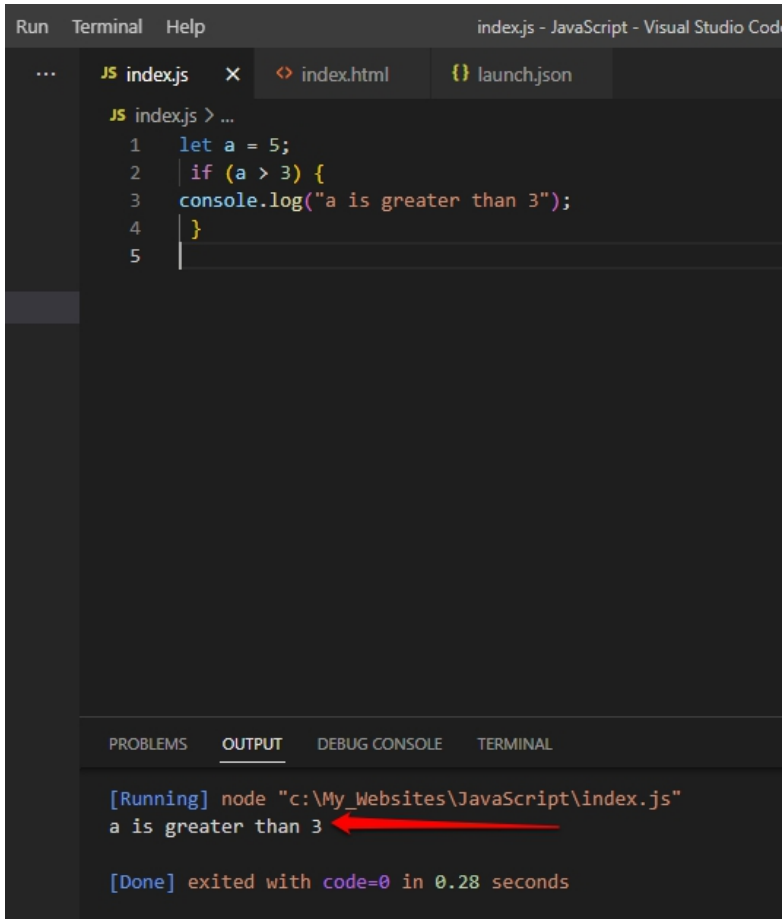
let a = 5;

if (a > 3) {

console.log("a is greater than 3");

}

In this example, the condition is **a > 3**, which evaluates to true because **a** is equal to 5, which is greater than 3. Therefore, the block of code inside the curly braces will be executed, and the message "a is greater than 3" will be printed on the console. The output of the code above is shown below.



The Else Statement:

The else statement can be used in conjunction with the if statement to execute a block of code if the condition is false. The else statement syntax is written as follows:

if (condition) {

// block of code to be executed if the condition is true

}

else {

// block of code to be executed if the condition is false

}

Here is an example:

let a = 2;

if (a > 3) {

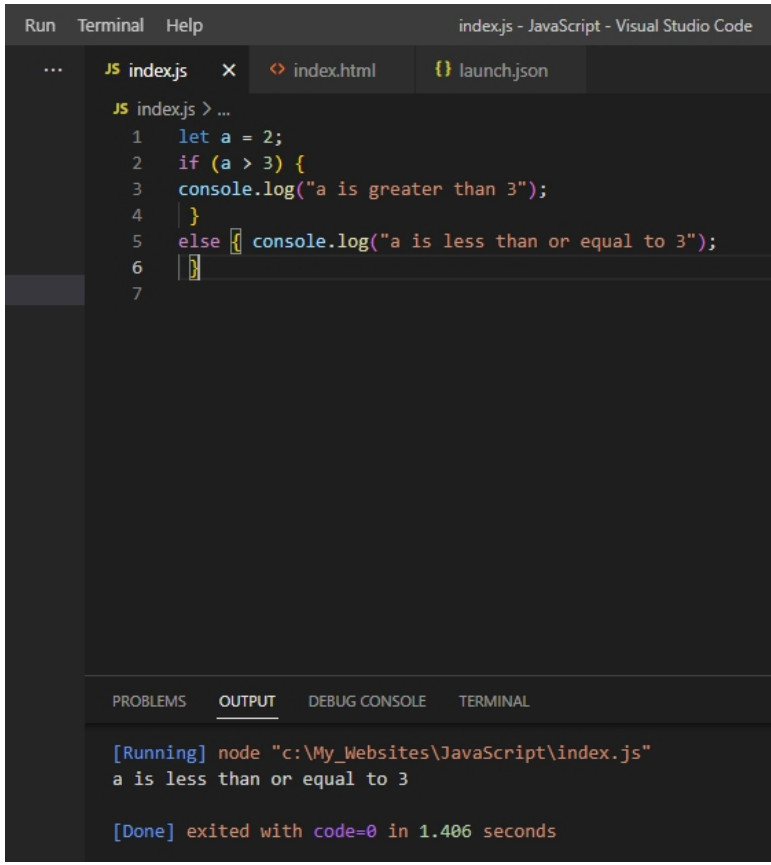
console.log("a is greater than 3");

}

else { console.log("a is less than or equal to 3");

}

In this example, the condition is **a > 3**, which evaluates to false because **a** is equal to 2, which is less than 3. Therefore, the block of code inside the else statement will be executed, and the message "a is less than or equal to 3" will be printed to the console. The output of the above code is shown below.



The else if statement:

The else if statement can be used in conjunction with the if statement to execute a block of code if a different condition is true. The else-if statement syntax is written as follows:

if (condition1) {

// block of code to be executed if condition1 is true

}

else if (condition2) {

// block of code to be executed if condition2 is true

}

else {

// block of code to be executed if all conditions are false

}

Here is an example:

let a = 7;

if (a > 10) {

console.log("a is greater than 10");

}

else if (a > 5) {

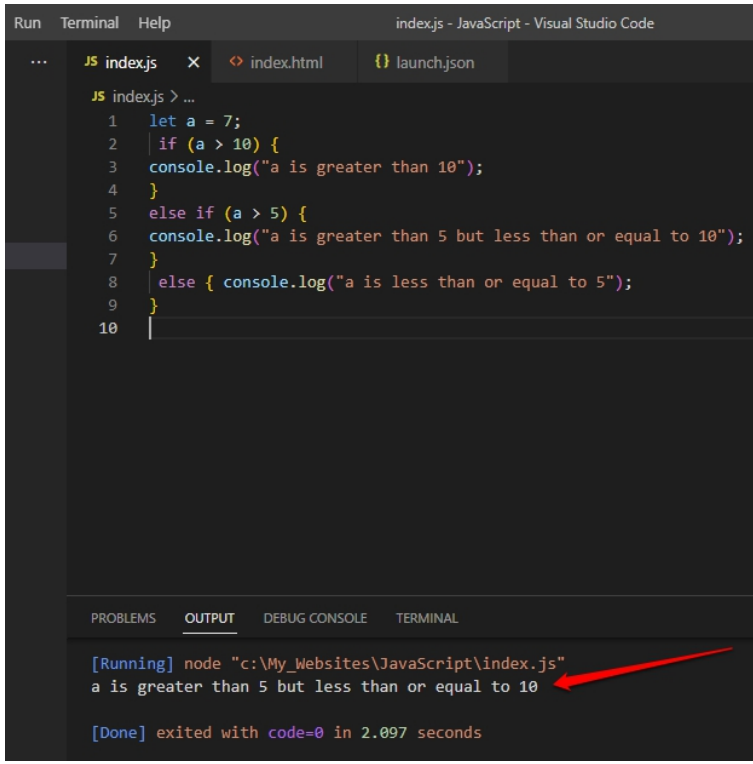
console.log("a is greater than 5 but less than or equal to 10");

}

else { console.log("a is less than or equal to 5");

}

In this example, condition **a > 10** is false, but condition **a > 5** is true, so the second block of code will be executed, and the message "a is greater than 5 but less than or equal to 10" will be printed to the console. The output of the code above is shown below.


Run Terminal Help
index.js - JavaScript - Visual Studio Code
index.js
index.html
launch.json
index.js > ...
1 let a = 7;
2 if (a > 10) {
3 console.log("a is greater than 10");
4 }
5 else if (a > 5) {
6 console.log("a is greater than 5 but less than or equal to 10");
7 }
8 else { console.log("a is less than or equal to 5");
9 }
10
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
[Running] node "c:\My_Websites\JavaScript\index.js"
a is greater than 5 but less than or equal to 10
[Done] exited with code=0 in 2.097 seconds

THE SWITCH STATEMENT

The switch statement is a control flow statement used in programming languages like JavaScript, Java, and C++. It allows a program to evaluate an expression and execute different actions based on the value of the expression.

The syntax for the switch statement in JavaScript is as follows:

```
switch(expression) {
  case value1:
    // code block
    break;
  case value2:
    // code block
    break;
```

```
  default:
    // code block
}
```

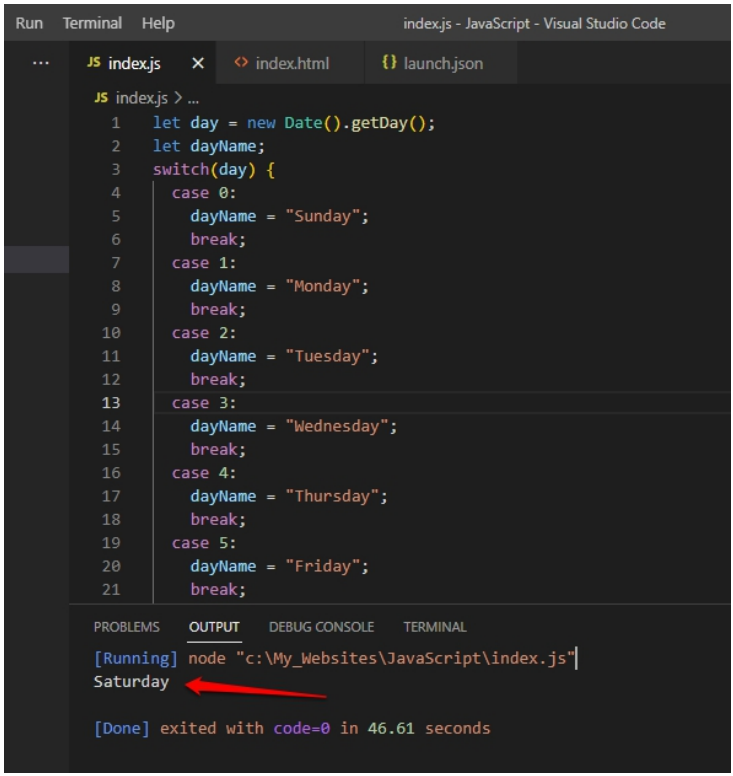
The expression is evaluated once, and the value is compared with the values of each case. If there is a match, the associated block of code is executed. If there is no match, the code in the default block is executed.

Here's an example of a switch statement in action:

```
let day = new Date().getDay();
let dayName;
switch(day) {
  case 0:
    dayName = "Sunday";
    break;
  case 1:
    dayName = "Monday";
    break;
  case 2:
    dayName = "Tuesday";
    break;
  case 3:
    dayName = "Wednesday";
    break;
  case 4:
    dayName = "Thursday";
    break;
  case 5:
```

```
    dayName = "Friday";
    break;
  case 6:
    dayName = "Saturday";
    break;
  default:
    dayName = "Invalid day";
}
console.log(dayName);
```

In this example, the switch statement evaluates the current day of the week using the **getDay()** method of the **Date** object. The code matches the day with the appropriate string value and assigns it to the **dayName** variable. If the day value does not match any of the cases, the default case assigns the **dayName** variable the value of "Invalid day". The result of the code above is shown in the screenshot below.



Switch statements are often used as an alternative to if-else statements when there are multiple conditions to check. They can also make code more readable and efficient in certain cases.

CHAPTER FIFTEEN

FUNCTIONS AND SCOPE

In JavaScript, functions are reusable blocks of code that can take in inputs (arguments) and return a value. They are a fundamental building block of programming and allow for the creation of modular and reusable code. In this chapter, we will cover the basics of declaring and calling functions, working with parameters and arguments, returning values, and understanding scope and closures.

DECLARING AND CALLING FUNCTIONS

Functions in JavaScript can be declared using the **function** keyword, followed by the name of the function and a set of parentheses. The function body is enclosed in curly braces **{}**. For instance:

function greet() {

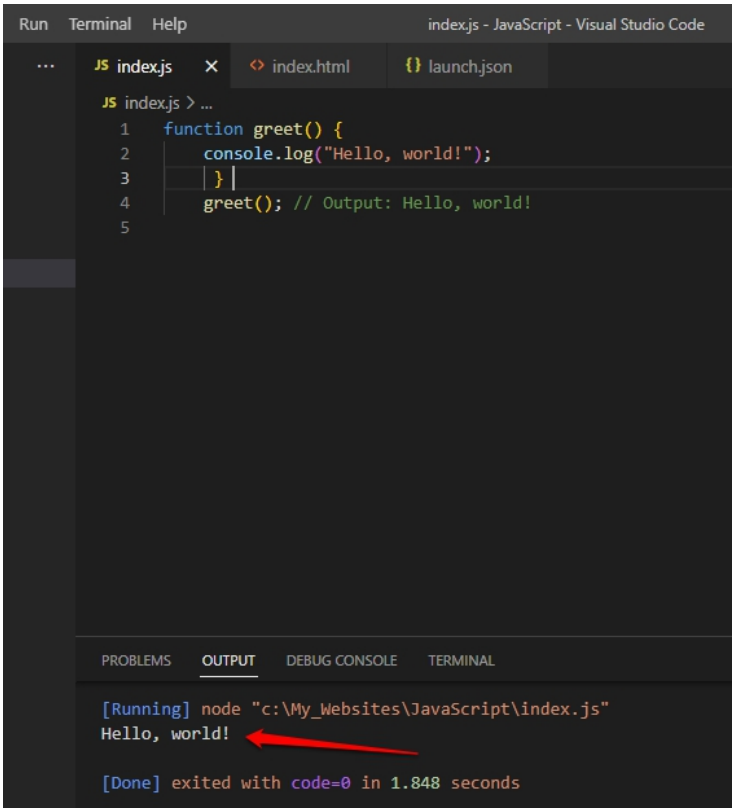
console.log("Hello, world!");

}

To call this function, simply use the function name followed by a set of parentheses:

greet(); // Output: Hello, world!

See the output below:



PARAMETERS AND ARGUMENTS

Functions can also take in inputs through parameters. Parameters are declared in the function declaration and are used to specify the expected inputs for the function. For example:

function greet(name) {

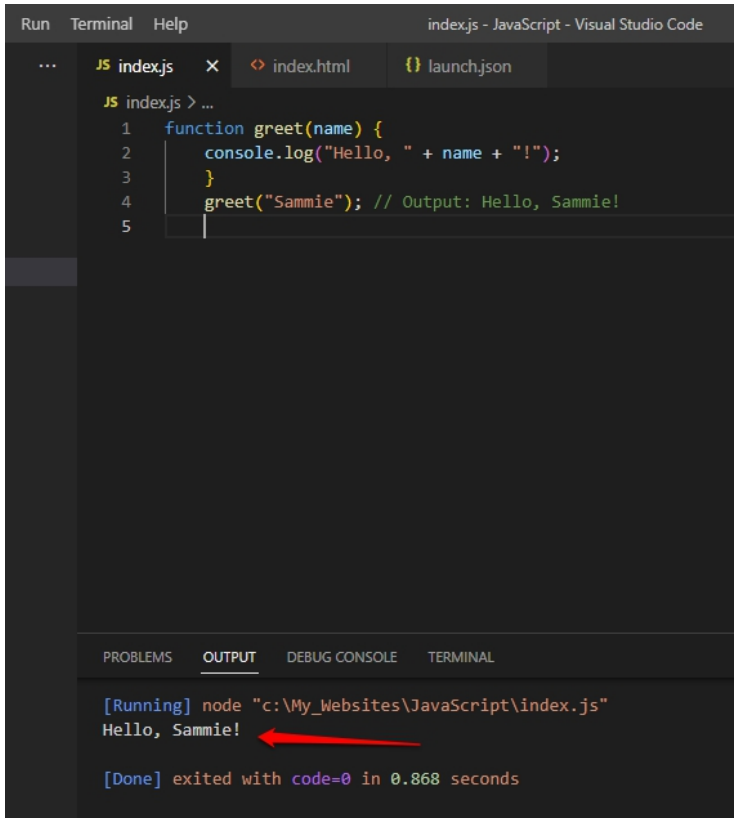
console.log("Hello, " + name + "!");

}

In the above example, the **greet** function takes in one parameter, **name**. To call the function with an argument, simply pass the value as an argument when calling the function:

greet("Sammie"); // Output: Hello, Sammie!

See the output below:



Functions can also take in multiple parameters, separated by commas:

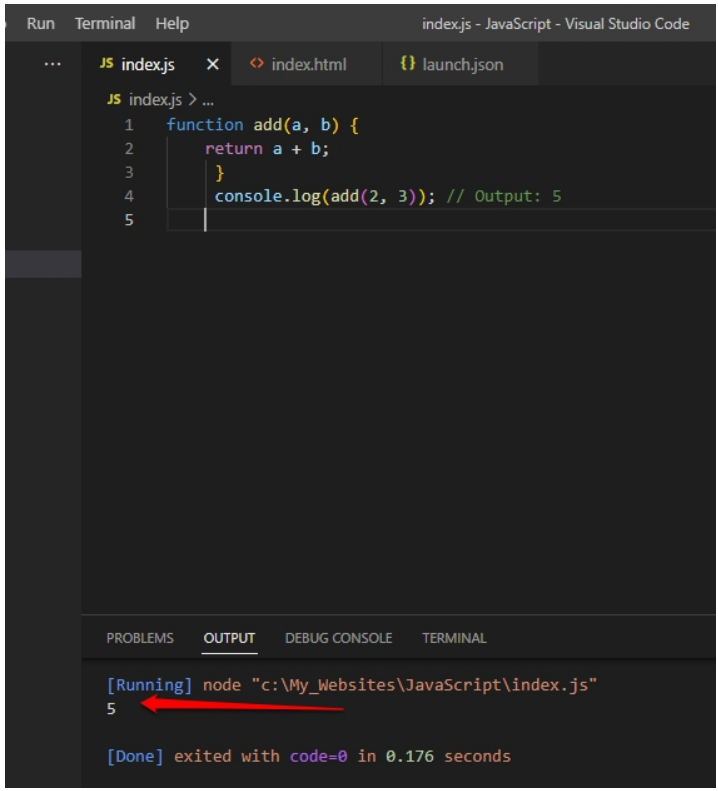
function add(a, b) {

return a + b;

}

console.log(add(2, 3)); // Output: 5

See the output below:



RETURN VALUES

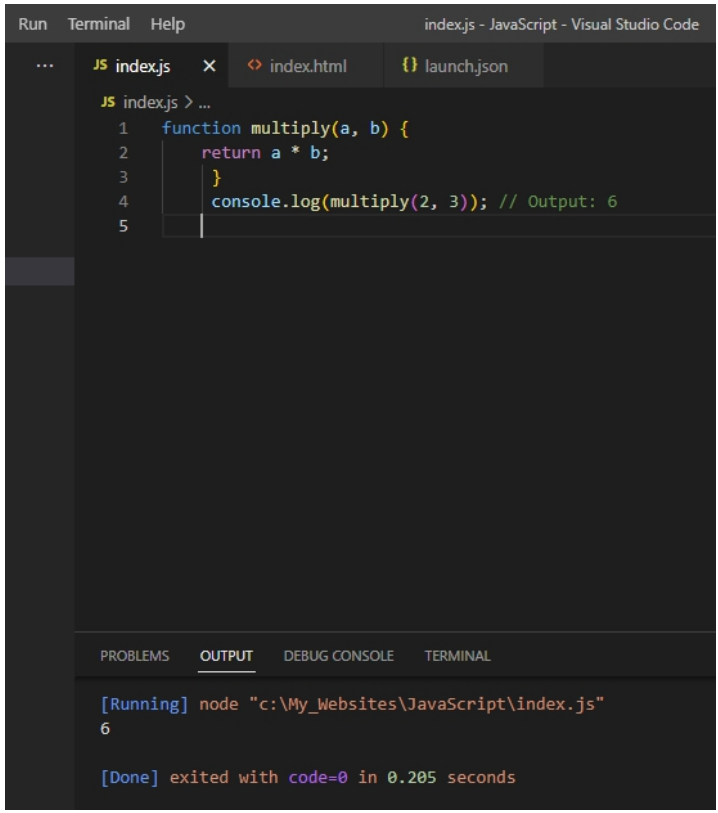
Return Values Functions can also return a value using the **return** keyword. This allows the function to provide a result that can be used in other parts of the code. For example:

function multiply(a, b) {

*return a * b; }*

console.log(multiply(2, 3)); // Output: 6

In the above example, the **multiply** function returns the product of **a** and **b**. See the output below.



SCOPE AND CLOSURES

Scope refers to the visibility of variables in different parts of the code. In JavaScript, there are two types of scope: global scope and local scope.

Global Scope

Global scope refers to variables that are declared outside of any function and can be accessed from anywhere in the code:

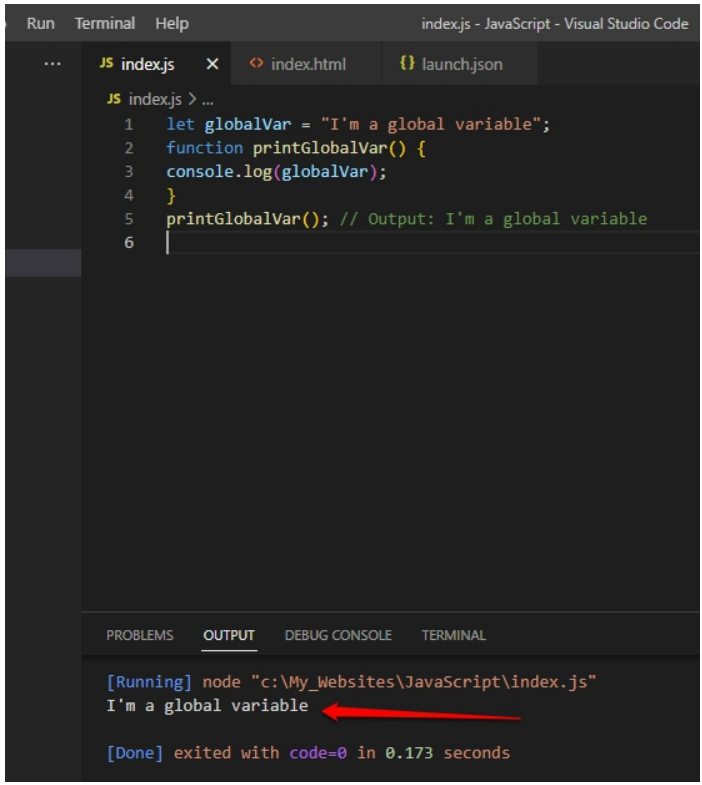
let globalVar = "I'm a global variable";

function printGlobalVar() {

console.log(globalVar); }

printGlobalVar(); // Output: I'm a global variable

See the output below:



Local Scope: Local scope refers to variables that are declared inside a function and can only be accessed within that function:

function printLocalVar() {

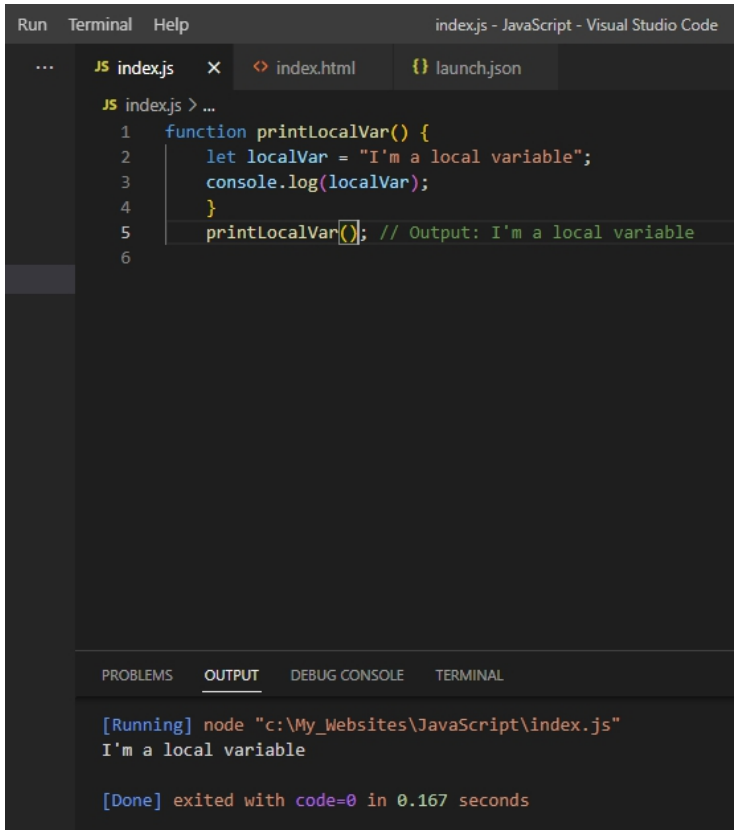
let localVar = "I'm a local variable";

console.log(localVar);

}

printLocalVar(); // Output: I'm a local variable

See the output below.



Closures

Closures refer to the ability of a function to access variables in its outer (enclosing) function, even after the outer function has returned. For example:

function outer() {

let x = 10;

function inner() {

console.log(x);

}

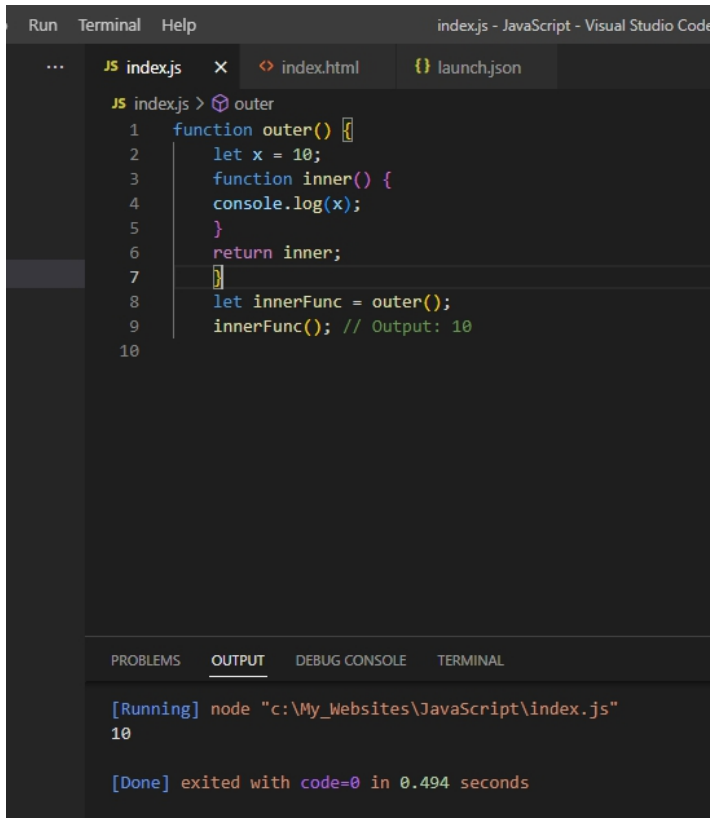
return inner;

}

let innerFunc = outer();

innerFunc(); // Output: 10

In the above example, the **outer** function returns the **inner** function, which has access to the **x** variable in the **outer** function's scope, even though the **outer** function has already returned. See the output below;



In the above example, the **outer** function returns the **inner** function, which has access to the **x** variable in the **outer** function's scope, even though the **outer** function has already returned.

Write extensively on this chapter with examples (3000 words).

CHAPTER SIXTEEN

ARRAYS AND LOOPS

Arrays and loops are two of the most fundamental concepts in JavaScript. Arrays are used to store collections of data, while loops are used to iterate over that data. In this chapter, we will dive into working with arrays, using array methods, and exploring different types of loops in JavaScript.

WORKING WITH ARRAYS AND ARRAY METHODS

Arrays are used to store collections of data in JavaScript. An array is a special type of variable that can hold multiple values. Each value in an array is called an element, and each element has a unique index number that represents its position in the array.

To declare an array, we use square brackets and separate the elements with commas. For example:

let fruits = ["apple", "banana", "orange"];

In the above example, we declared an array named **fruits** with three elements: "apple", "banana", and "orange".

We can access individual elements of an array using their index numbers. The index of the first element in an array is always 0. For example, to access the first element of the **fruits** array, we would use:

let firstFruit = fruits[0];

This would assign the value "apple" to the **firstFruit** variable.

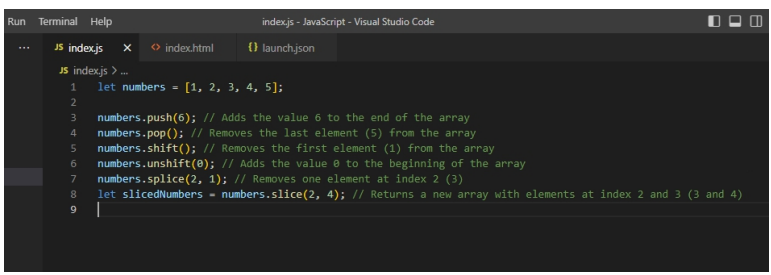
Arrays also have several built-in methods that allow us to manipulate the data they contain. Some of the most commonly used array methods include:

- **push()**: Adds one or more elements to the end of an array.
- **pop()**: Removes the last element from an array and returns it.
- **shift()**: Removes the first element from an array and

returns it.

- **unshift()**: Adds one or more elements to the beginning of an array.
- **splice()**: Adds or removes elements from an array at a specified index.
- **slice()**: Returns a new array containing a portion of the original array.

Here's an example that demonstrates how to use some of these methods:


Run Terminal Help
index.js - JavaScript - Visual Studio Code
index.js
index.html
launch.json
let numbers = [1, 2, 3, 4, 5];
numbers.push(6); // Adds the value 6 to the end of the array
numbers.pop(); // Removes the last element (5) from the array
numbers.shift(); // Removes the first element (1) from the array
numbers.unshift(0); // Adds the value 0 to the beginning of the array
numbers.splice(2, 1); // Removes one element at index 2 (3)
let slicedNumbers = numbers.slice(2, 4); // Returns a new array with elements at index 2 and 3 (3 and 4)

The For Loop

The for loop is used to iterate over an array or any other type of iterable object. It has three parts: initialization, condition, and update. Here's an example of a for loop that iterates over an array of numbers and prints each one to the console:

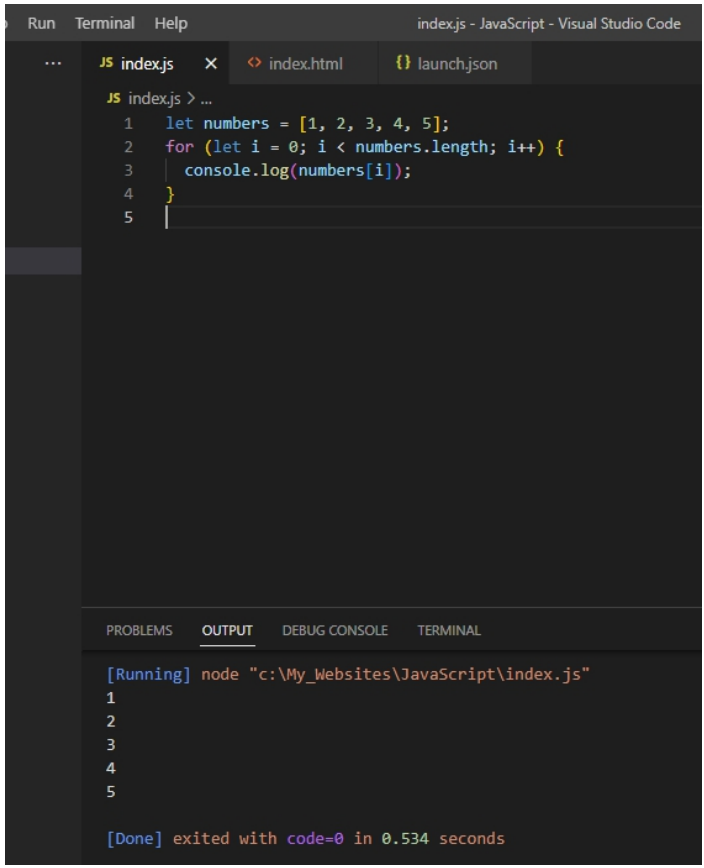
let numbers = [1, 2, 3, 4, 5];

for (let i = 0; i < numbers.length; i++) {

console.log(numbers[i]);

}

In the above example, we use the **length** property of the **numbers** array to determine the number of iterations needed. The loop starts at 0 and continues until the index is equal to the length of the array. See the output below.



The While Loop

The while loop is a control flow statement that allows you to execute a block of code repeatedly while a specific condition is true. The syntax for the while loop is as follows:

while (condition) {

// code to be executed

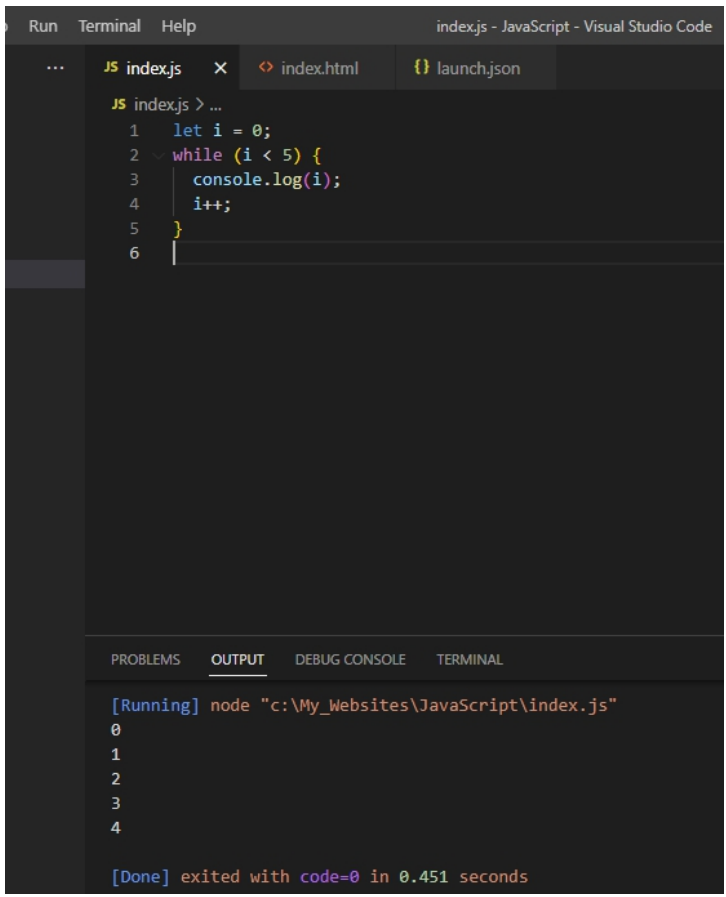
}

Here, the **condition** is a Boolean expression that is evaluated before each iteration of the loop. If the **condition** is true, the code block is executed. This process is repeated until the **condition** is false.

Let's take an example to understand how the while loop works:

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

In this example, the while loop will execute the code block until **i** is less than 5. The output of the above code will be:



The Do-While Loop

The do-while loop is similar to the while loop but with one significant difference: the code block is executed at least once, even

if the condition is false. The syntax for the do-while loop is as follows:

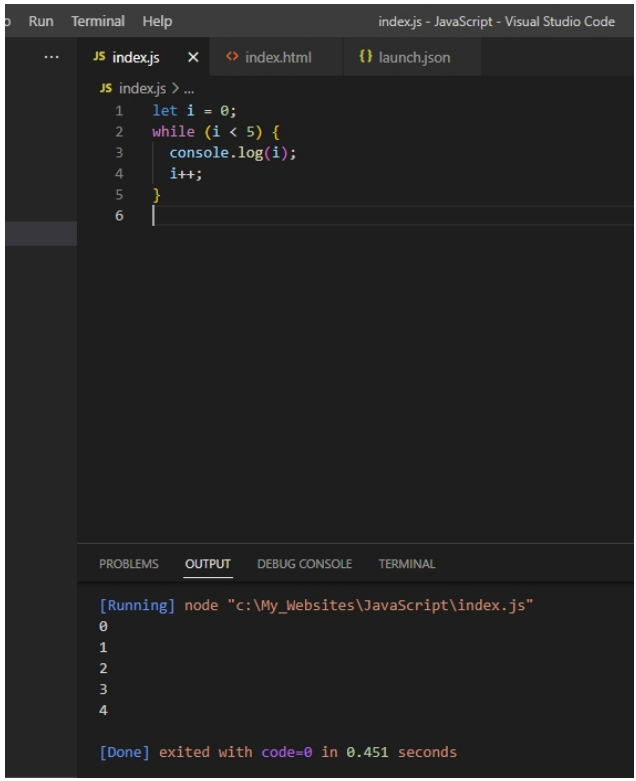
```
while (condition) {
   // code to be executed
}
```

Here, the **condition** is a Boolean expression that is evaluated after each iteration of the loop. If the **condition** is true, the code block is executed again. This process is repeated until the **condition** is false.

Let's take an example to understand how the do-while loop works:

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

In this example, the code block will execute at least once, even if **i** is not less than 5. The output of the above code will be:



The ForEach Method

The forEach method is a built-in function in JavaScript that allows you to iterate over an array and execute a function on each element. The syntax for the forEach method is as follows:

array.forEach(function(currentValue, index, array) {

// code to be executed

});

Here, currentValue denotes the element currently being processed, index denotes the element's index, and array denotes the array is scanned. Each element of the array receives an execution of the code block.

To further comprehend how the forEach method functions, let's look at an example:

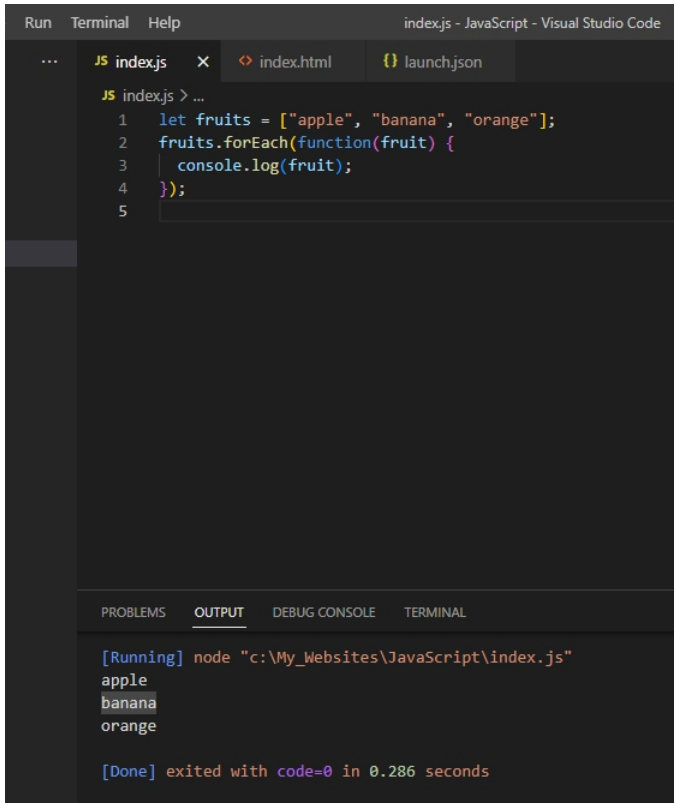
let fruits = ["apple", "banana", "orange"];

fruits.forEach(function(fruit) {

* console.log(fruit);*

});

In this example, the forEach method will execute the code block for each element of the **fruits** array. The output of the above code will be:

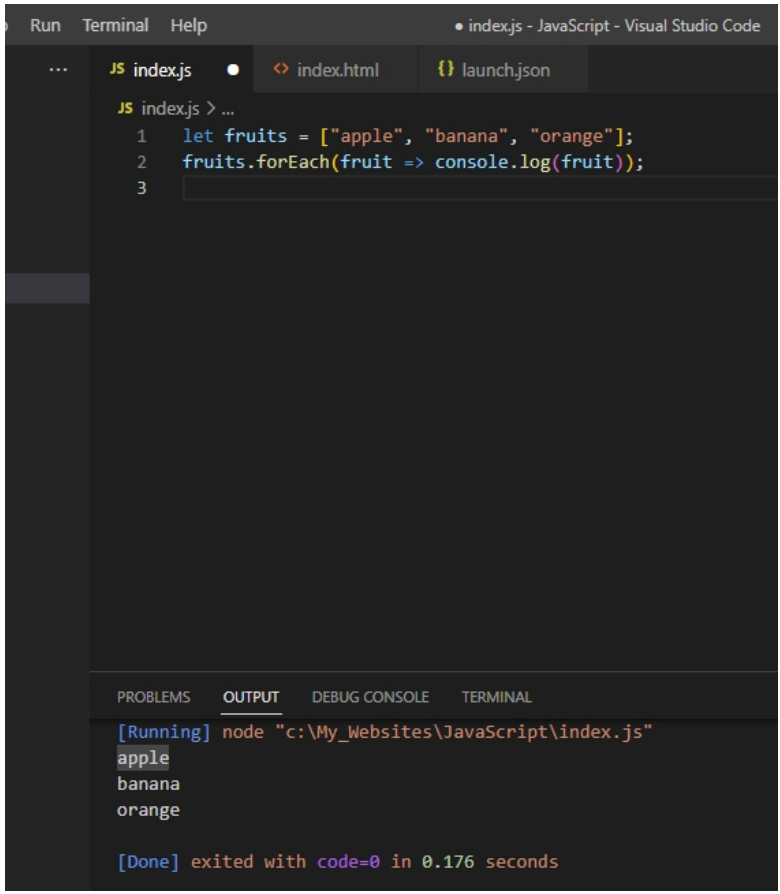


To make the code shorter, you may also use the arrow function syntax:

let fruits = ["apple", "banana", "orange"];

fruits.forEach(fruit => console.log(fruit));

See the output below:



CHAPTER SEVENTEEN

OBJECTS AND CLASSES

An object in JavaScript is a group of properties, each of which is a key-value pair. Objects may have characteristics that are either other objects or primitive data types (such as texts, integers, and booleans). Additionally, objects are capable of having methods, which are defined as the object's properties and functions. Because they make it possible for programmers to efficiently organize and manage code, objects are crucial to JavaScript programming.

CREATING AND WORKING WITH OBJECTS

To create an object in JavaScript, you can use either the object literal notation or the Object constructor. The object literal notation is a shorthand way of creating an object by enclosing a list of key-value pairs in curly braces. For example:

```
let person = {
  name: "John",
  age: 30,
  city: "New York"
};
```

In this example, we create an object, called a person with three properties: name, age, and city.

You can also create an object using the Object constructor. The Object constructor creates an empty object that you can add properties and methods to. For example:

```
let person = new Object();
person.name = "John";
person.age = 30;
person.city = "New York";
```

This code creates an empty object called a person using the new keyword and then adds three properties to it using dot notation.

Object Methods

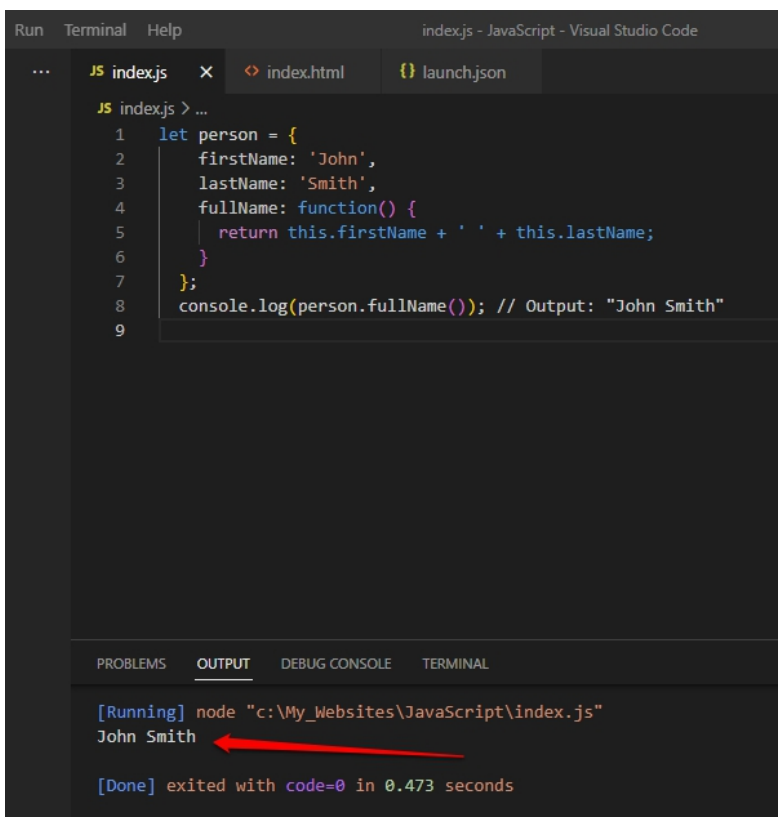
Objects in JavaScript can also have methods, which are functions that are defined as properties of the object. Methods can access the object's properties and other methods. In JavaScript, objects can have methods, which are functions that belong to the object and can operate on its properties. To create a method for an object, we define a function as a property of the object.

Here's an example:

```
let person = {
```

```
  firstName: 'John',
  lastName: 'Smith',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.fullName()); // Output: "John Smith"
```

In this example, we define an object called a person with properties firstName, lastName, and a method fullName. The fullName method concatenates the firstName and lastName properties and returns the result. See the output below.

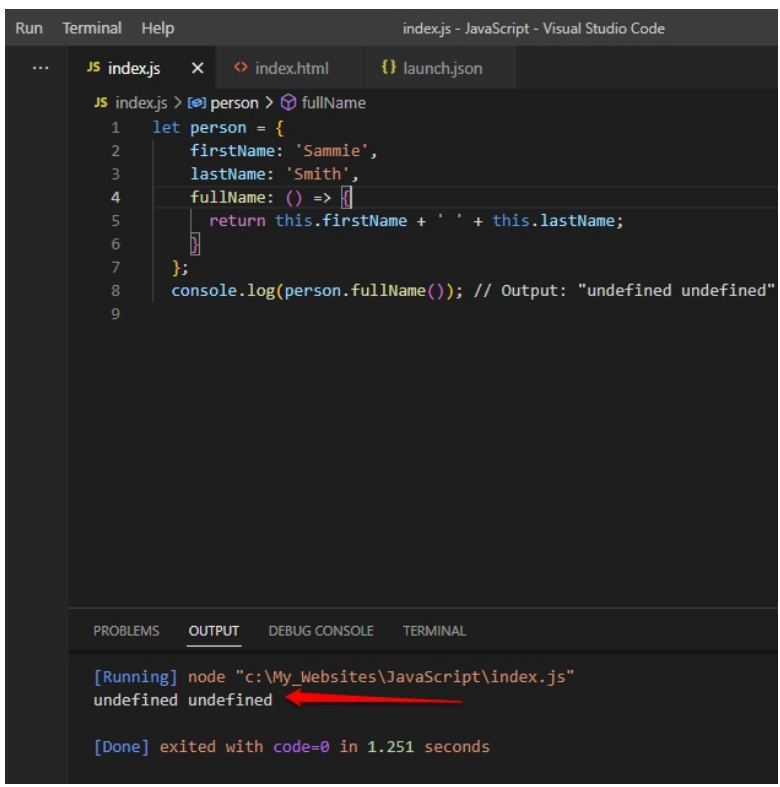


We can also define object methods using ES6 syntax with arrow

functions:

```
let person = {
  firstName: 'Sammie',
  lastName: 'Smith',
  fullName: () => {
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.fullName()); // Output: "undefined undefined"
```

See the output below.

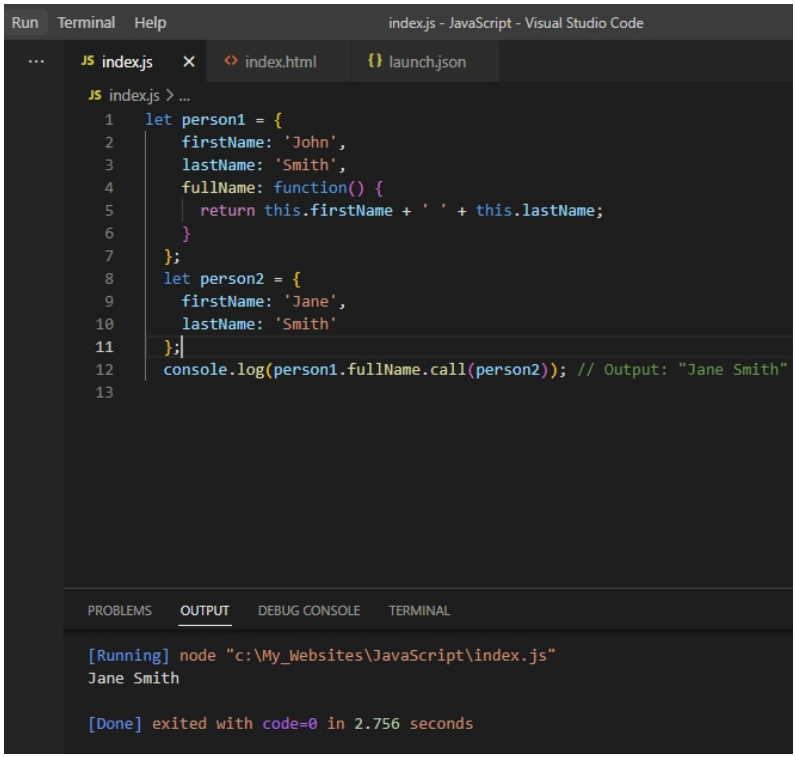


However, using arrow functions for object methods is generally not recommended, as it can cause issues with this keyword.

Object methods can also be called outside of the object context by using the call() or apply() method. For example:

```
let person1 = {
  firstName: 'John',
  lastName: 'Smith',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};
let person2 = {
  firstName: 'Jane',
  lastName: 'Smith'
};
console.log(person1.fullName.call(person2)); // Output: "Jane Smith"
```

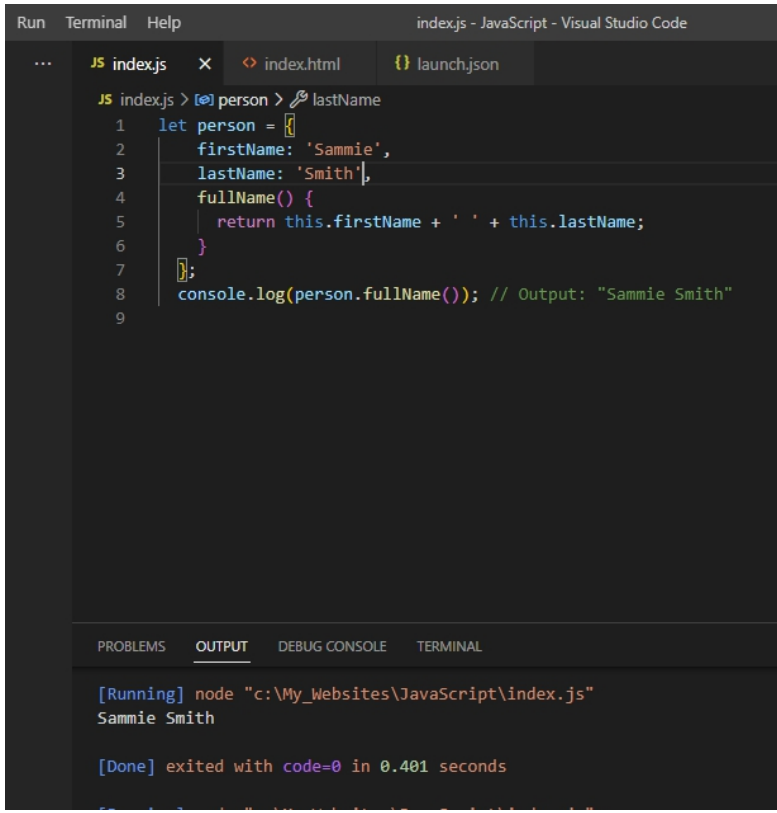
In this example, we define two objects person1 and person2. person1 has a fullName method, while person2 does not. We can still use the call() method to call person1's fullName method with person2 as its this value, effectively "borrowing" the method. See the screenshot of the output below.



Object methods can also be defined using shorthand notation:

```
let person = {
  firstName: 'Sammie',
  lastName: 'Smith',
  fullName() {
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.fullName()); // Output: "Sammie Smith"
```

In this example, we define the fullName method using a shorthand notation. The result is the same as the previous example. See the screenshot of the output below.



CREATING CLASSES

In addition to creating objects using the object literal notation or the Object constructor, you can also create objects using classes. A class is a blueprint for creating objects, and it defines a set of properties and methods that the objects will have. To define a class in JavaScript, you use the class keyword, followed by the name of the class. For example:

In JavaScript, you can create classes using the class keyword. Classes are a way to define blueprints for objects that you want to create. They provide a way to encapsulate data and behavior in a single entity. A class can have properties and methods just like an object.

Here's an example of how to create a class in JavaScript:

class Person {

```
  constructor(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
  canVote() {
    return this.age >= 18;
  }
}
```

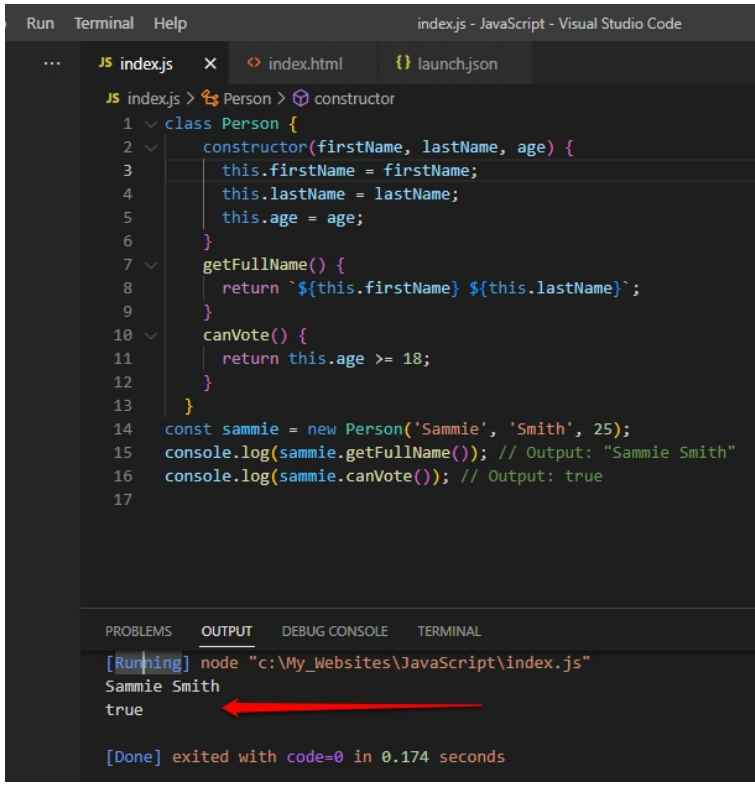
In this example, we define a Person class with a constructor that takes in three parameters: firstName, lastName, and age. The constructor assigns these values to the properties of the object being created.

The class also has two methods: getFullName and canVote. getFullName returns the full name of the person by concatenating the first and last names, while canVote checks if the person is 18 years or older and returns true or false.

To create an object from this class, you use the new keyword and call the constructor with the necessary arguments:

```
const sammie = new Person('Sammie', 'Smith', 25);
console.log(john.getFullName()); // Output: "Sammie Smith"
console.log(sammie.canVote()); // Output: true
```

Here, we create a new Person object named sammie with the first name "Sammie", last name "Smith", and age 25. We then call the getFullName and canVote methods on the sammie object. See the screenshot of the output below.



Classes in JavaScript also support inheritance, which allows you to create a new class based on an existing one. You can use the extends keyword to create a subclass that inherits properties and methods from its parent class.

For example, we can create a Student class that extends the Person class and adds a new property, major:

```
class Person {
    constructor(firstName, lastName, age) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
    }
      getFullName() {
```

```
    return `${this.firstName} ${this.lastName}`;
  }
    canVote() {
    return this.age >= 18;
  }
}
  class Student extends Person {
  constructor(firstName, lastName, age, major) {
    super(firstName, lastName, age);
    this.major = major;
  }
    getMajor() {
    return this.major;
  }
}
```

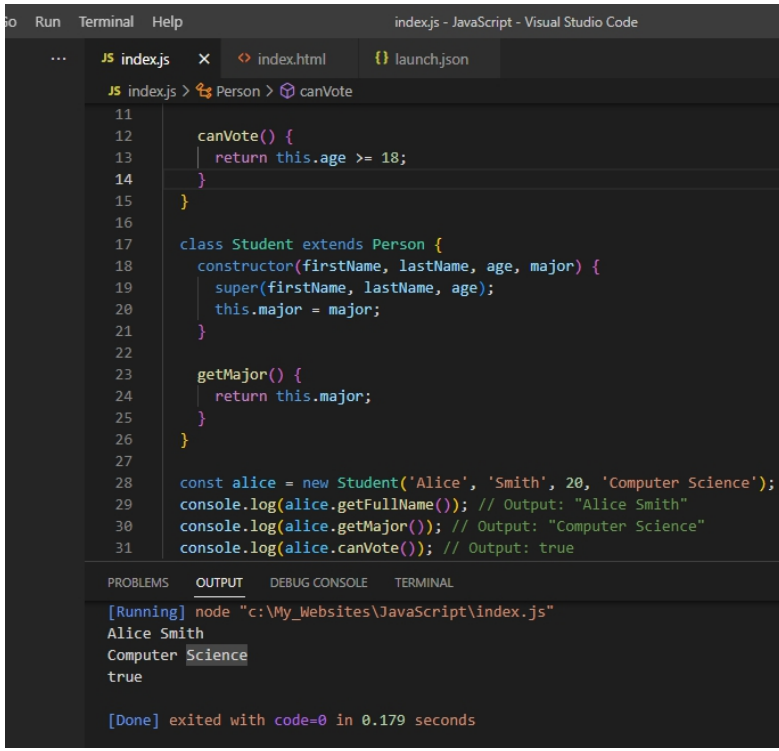
Here, we define a Student class that extends the Person class using the extends keyword. The constructor takes four parameters: firstName, lastName, age, and major. We call the super method with the necessary arguments to invoke the constructor of the parent class and inherit its properties.

The Student class also has a new method called getMajor that returns the value of the major property. To create a Student object, you can use the same syntax as creating a Person object:

```
const alice = new Student('Alice', 'Smith', 20, 'Computer Science');
  console.log(alice.getFullName()); // Output: "Alice Smith"
  console.log(alice.getMajor()); // Output: "Computer Science"
  console.log(alice.canVote()); // Output: true
```

Here, we create a new Student object named alice with the first

name "Alice", last name "Smith", age 20, and major "Computer Science". We call the getFullName, getMajor, and canVote methods on the alice object. See the output of the code above below.



CONSTRUCTORS AND PROTOTYPES

Constructors and prototypes are an essential part of object-oriented programming in JavaScript. In this section, we will explore how constructors and prototypes work and how they can be used to create objects and classes.

Constructors are functions that are used to create new objects. They define the properties and methods of the object and can also define default values for these properties. When a constructor is called with the new keyword, a new object is created, and the constructor initializes the object's properties.

Here's an example of a constructor function:

function Person(name, age) {

```
  this.name = name;
  this.age = age;
}
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

In this example, we have created a constructor function called Person that takes two arguments, name, and age. Inside the constructor, we set the name and age properties of the object using this keyword. We then create two new Person objects, person1, and person2, using the new keyword and passing in the name and age arguments.

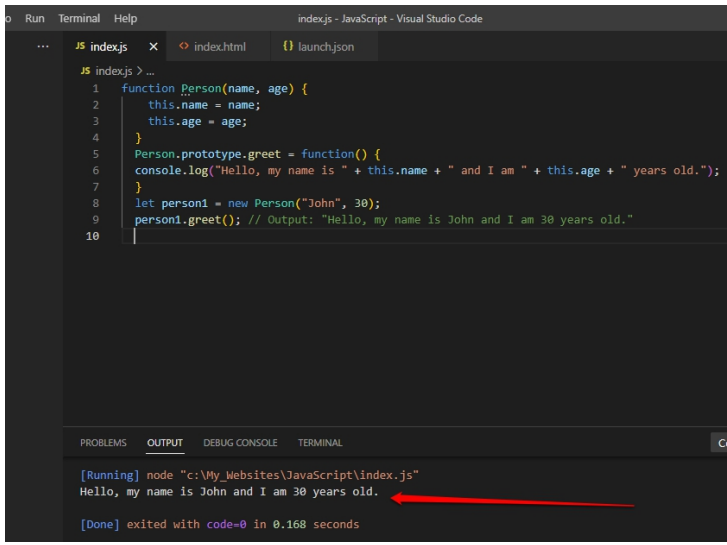
Prototypes are used to add new properties and methods to an object or class. They allow us to define a set of properties and methods that can be shared by all objects created from the same constructor function.

Here's an example of how to add a method to a constructor's prototype:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.greet = function() {
console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");
}
let person1 = new Person("John", 30);
person1.greet(); // Output: "Hello, my name is John and I am 30 years old."
```

In this example, we have added a new method called greet to the Person constructor's prototype. This method is then available to all objects created from the Person constructor. We create a new

Person object, person1, and call the greet method on it to output a message to the console. See the screenshot below.


Run Terminal Help
index.js - JavaScript - Visual Studio Code
index.js
index.html
launch.json
function Person(name, age) {
this.name = name;
this.age = age;
}
Person.prototype.greet = function() {
console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");
}
let person1 = new Person("John", 30);
person1.greet(); // Output: "Hello, my name is John and I am 30 years old."
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
[Running] node "c:\My_Websites\JavaScript\index.js"
Hello, my name is John and I am 30 years old.
[Done] exited with code=0 in 0.168 seconds

Another use of prototypes is to create classes in JavaScript. A class is a blueprint for creating objects that share the same properties and methods. We can use a constructor function and its prototype to create a class.

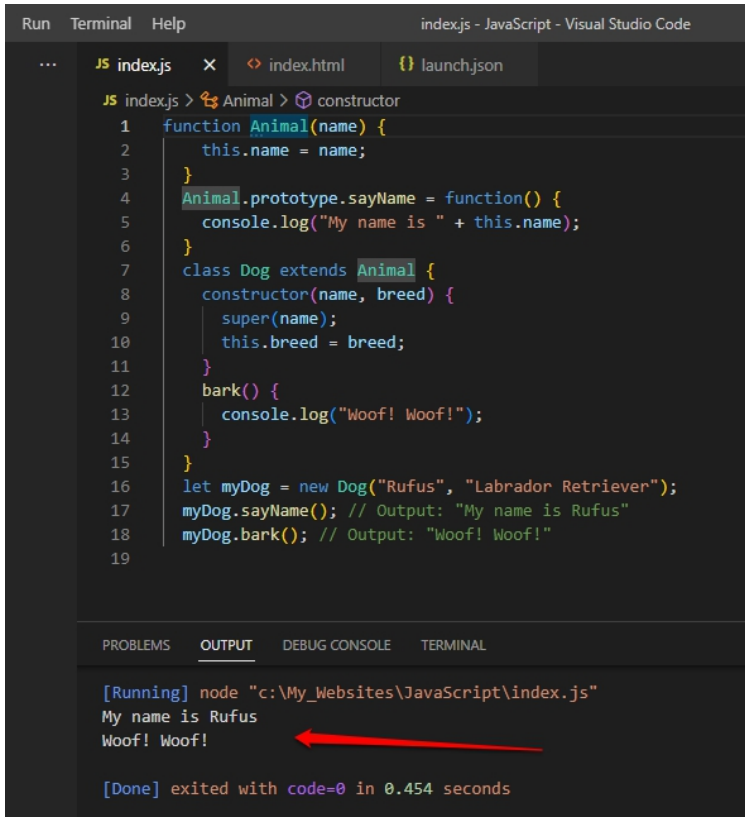
Here's an example of how to create a class using a constructor and its prototype:

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.sayName = function() {
  console.log("My name is " + this.name);
}
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
```

```
  }
  bark() {
    console.log("Woof! Woof!");
  }
}
let myDog = new Dog("Rufus", "Labrador Retriever");
myDog.sayName(); // Output: "My name is Rufus"
myDog.bark(); // Output: "Woof! Woof!"
```

In this example, we create a new class called Dog that extends the Animal class. We use the constructor function and its prototype to define the properties and methods of the Dog class. We also use the super keyword to call the Animal constructor and set the name property. We then define a new method, bark, that is specific to the Dog class.

We create a new Dog object, myDog, and call the sayName and bark methods on it to output messages to the console. See the output below.



CHAPTER EIGHTEEN

DOM MANIPULATION

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. In this chapter, we will explore the basics of DOM manipulation using JavaScript.

INTRODUCTION TO THE DOCUMENT OBJECT MODEL (DOM)

When a web page is loaded, the browser creates a Document Object Model (DOM) of the page. The DOM is an object-oriented representation of the web page, which can be modified with a scripting language such as JavaScript.

The DOM consists of a hierarchical tree of nodes. Each node represents an object in the document, such as an element, attribute, or text node. The DOM tree can be traversed and manipulated using JavaScript.

SELECTING AND MANIPULATING ELEMENTS

One of the most common tasks when working with the DOM is selecting and manipulating elements. This can be done using various methods provided by the DOM API.

getElementById

The getElementById method is used to select an element with a specific ID. It returns a reference to the element if it exists in the DOM, or null if it does not.

For example, let's say we have an HTML element with an ID of "myElement". We can select this element using the following code:

const element = document.getElementById("myElement");

querySelector

The querySelector method is used to select an element that matches a specified CSS selector. It returns the first element that matches the selector, or null if no element matches.

For example, let's say we have an HTML element with a class of "myClass". We can select this element using the following code:

const element = document.querySelector(".myClass");

querySelectorAll

The querySelectorAll method is used to select all elements that match a specified CSS selector. It returns a NodeList object, which can be iterated over using a for loop.

For example, let's say we have several HTML elements with a class of "myClass". We can select all of these elements using the following code:

const elements = document.querySelectorAll(".myClass");

MANIPULATING ELEMENTS

Once we have selected an element, we can manipulate it using various properties and methods provided by the DOM API.

innerHTML

The innerHTML property is used to get or set the HTML content of an element. For example, let's say we have an HTML element with an ID of "myElement". We can change its content using the following code:

const element = document.getElementById("myElement");

element.innerHTML = "<h1>Hello World!</h1>";

textContent

The textContent property is used to get or set the text content of an element. It returns the text content of all child nodes, excluding HTML tags. For example, let's say we have an HTML element with an ID of "myElement". We can change its text content using the following code:

const element = document.getElementById("myElement");

element.textContent = "Hello World!";

setAttribute

The setAttribute method is used to set the value of an attribute for an element. For example, let's say we have an HTML element with an ID of "myElement". We can set its "src" attribute using the following code:

const element = document.getElementById("myElement");

element.setAttribute("src", "image.jpg");

EVENT HANDLING

Event handling is a critical aspect of web development that enables web applications to respond to user interactions. The user can interact with a web page by clicking buttons, typing into text fields, hovering over elements, scrolling, and more. The DOM allows us to

manipulate elements on a web page, and with event handling, we can create interactive web applications that can respond to user interactions.

In JavaScript, an event is an action that occurs in the browser, such as clicking a button, submitting a form, or scrolling the page. An event listener is a function that listens for a specific event to occur and performs an action when the event is detected.

There are different ways to attach event listeners to HTML elements. The most common way is to use the addEventListener() method. This method takes two arguments: the name of the event to listen for and the function that should be executed when the event occurs. Here is an example of how to add an event listener to a button that logs a message to the console when it is clicked:

```
const myButton = document.querySelector('#myButton');

myButton.addEventListener('click', function() {

  console.log('Button clicked!');

});
```

In this example, we first use the querySelector() method to select the button element with the ID of myButton. We then use the addEventListener() method to attach a click event listener to the button. When the button is clicked, the function passed as the second argument to addEventListener() will be executed and the message "Button clicked!" will be logged to the console.

We can also use the removeEventListener() method to remove an event listener from an element. This method takes the same arguments as addEventListener(): the name of the event to remove and the function that was originally used as the event listener.

```
const myButton = document.querySelector('#myButton');

function handleClick() {

  console.log('Button clicked!');

}

myButton.addEventListener('click', handleClick);
```

// Later, we can remove the event listener

myButton.removeEventListener('click', handleClick);

In this example, we first define a function handleClick() that logs a message to the console when called. We then use addEventListener() to attach handleClick() as the event listener for the button's click event. Finally, we use removeEventListener() to remove the event listener.

Besides addEventListener() and removeEventListener(), some other methods and properties are commonly used when working with events. Some of these include:

- **event.target:** This property returns the element that triggered the event.
- **event.preventDefault():** This method prevents the default behavior of an element. For example, calling event.preventDefault() on a form prevents it from submitting when the submit button is clicked.
- **event.stopPropagation():** This method stops the propagation of an event. For example, calling event.stopPropagation() on a child element prevents the event from propagating to its parent elements.

Here is an example of how to use some of these methods:

```
const myButton = document.querySelector('#myButton');

myButton.addEventListener('click', function(event) {
  console.log('Button clicked!');
  console.log('Target:', event.target);
  event.preventDefault();
  event.stopPropagation();
});
```

In this example, we attach a click event listener to the button. When the button is clicked, the function passed as the second argument to addEventListener() will be executed. Inside this function, we log a message to the console, as well as the target element that triggered

the event using the event.target property. We also call event.preventDefault() and event.stopPropagation() to prevent the default behavior of the button and stop the event from propagating to its parent elements.

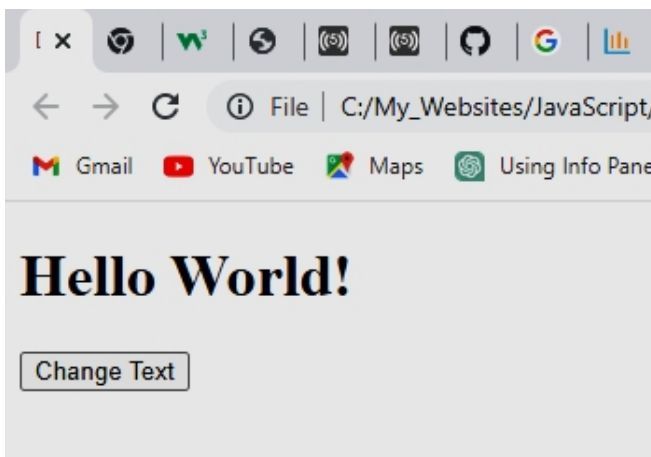
PRACTICAL IMPLEMENTATION OF DOM MANIPULATION

Below is an example of DOM Manipulation implementation

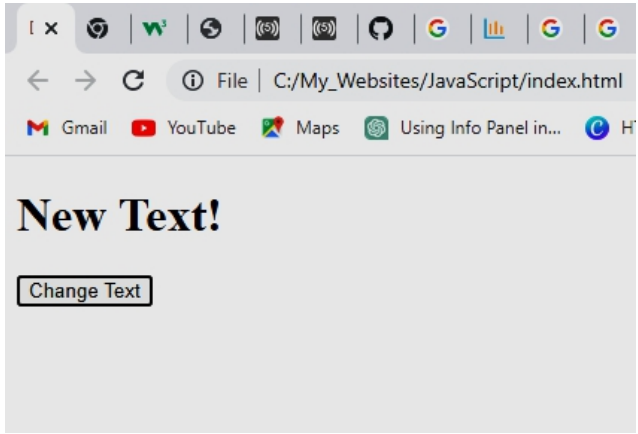
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Manipulation Example</title>
</head>
<body>
  <h1 id="my-heading">Hello World!</h1>
  <button id="my-button">Change Text</button>
  <script src="app.js"></script>
</body>
<Script>
    // select the button and heading elements
const button = document.getElementById('my-button');
const heading = document.getElementById('my-heading');
// add an event listener to the button
button.addEventListener('click', () => {
```

```
  // change the text of the heading element
  heading.textContent = 'New Text!';
});
</Script>
</html>
```

In this example, we first select the button and heading elements using their id attributes using the document.getElementById() method. Then, we add an event listener to the button using the addEventListener() method, which listens for a "click" event on the button. When the button is clicked, the anonymous arrow function inside the event listener is executed, which changes the text of the heading element using the textContent property. See the screenshots below.



After clicking on Change text, you should have the result below.



This is just a simple example, but DOM manipulation can be used to do many things on a webpage, such as changing CSS styles, creating and removing HTML elements, and more.

CHAPTER NINETEEN

ASYNCHRONOUS JAVASCRIPT

Asynchronous programming is a technique used in JavaScript to execute code that takes a long time to complete. JavaScript code typically executes in a single-threaded environment, which means that long-running code can cause delays in other parts of the program. Asynchronous programming allows us to avoid these delays by executing long-running code in the background while continuing to execute other code.

In this chapter, we'll explore the different methods of asynchronous programming in JavaScript, including callback functions, promises,

and async/await.

UNDERSTANDING ASYNCHRONOUS PROGRAMMING

In synchronous programming, code is executed linearly, one instruction at a time. In contrast, asynchronous programming allows code to be executed out of order, with different parts of the program executing simultaneously.

In JavaScript, asynchronous programming is typically used when dealing with operations that take a long time to complete, such as network requests, file system operations, or database queries.

Asynchronous programming is a programming paradigm that allows the code to continue executing while waiting for a long-running task to finish. It's commonly used for operations that take a lot of time to complete, like network requests or file input/output. In contrast to synchronous programming, where the code blocks and waits for the task to complete before continuing, asynchronous programming lets the code proceed without waiting for the task to finish.

The benefit of asynchronous programming is that it can improve the performance and responsiveness of an application. Without it, the application would have to wait for each long-running task to finish before proceeding to the next, making the application appear slow and unresponsive. By using asynchronous programming, the application can continue to respond to user input while performing these long-running tasks in the background.

One common example of asynchronous programming is a network request. When a user clicks a button to fetch data from a server, the application can use an asynchronous function to send the request and continue executing code while waiting for a response. When the response arrives, the function can be notified and process the data.

However, asynchronous programming can be complex and difficult to manage, particularly when multiple asynchronous tasks are running simultaneously. That's why JavaScript provides several tools to simplify the process, including callback functions, promises,

and async/await.

CALLBACK FUNCTIONS

Callback functions are functions that are passed as arguments to other functions and are executed when certain events occur. These events can be anything from a user clicking on a button to an HTTP request returning a response. Callback functions are a fundamental concept in asynchronous programming in JavaScript.

Let's look at an example of a callback function in action. Suppose we have an array of numbers and we want to iterate over the array and double each number. We can use the map method to achieve this:

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(function(number) {
  return number * 2;
});

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

In this example, we're passing a callback function to the map method. The map method will execute this function for each element in the numbers array and return a new array with the transformed values.

Let's look at another example where we use a callback function to handle an asynchronous operation. Suppose we have a function that makes an HTTP request to an API and returns the response data. We want to be able to do something with this data once it's returned, but we don't know exactly when the data will be returned. We can use a callback function to handle this:

```
function makeHttpRequest(url, callback) {
  const xhr = new XMLHttpRequest();
    xhr.addEventListener('load', function() {
    if (xhr.status === 200) {
      callback(xhr.responseText);
```

```
    }
  });
    xhr.open('GET', url);
  xhr.send();
}
makeHttpRequest('https://jsonplaceholder.typicode.com/posts/1',
function(responseData) {
  const data = JSON.parse(responseData);
  console.log(data);
});
```

In this example, we're passing a callback function as the second argument to the makeHttpRequest function. Once the HTTP request is complete and we have the response data, we're calling the callback function with the data as the argument. In this case, we're simply logging the data to the console, but we could do anything with the data inside the callback function.

Callback functions can also be used to handle errors that occur during an asynchronous operation. Let's modify our makeHttpRequest function to handle errors with a callback function:

```
function makeHttpRequest(url, successCallback, errorCallback) {
  const xhr = new XMLHttpRequest();
    xhr.addEventListener('load', function() {
    if (xhr.status === 200) {
      successCallback(xhr.responseText);
    } else {
      errorCallback(xhr.statusText);
    }
  });
```

```
    xhr.addEventListener('error', function() {
    errorCallback('Network Error');
  });
    xhr.open('GET', url);
  xhr.send();
}
makeHttpRequest('https://jsonplaceholder.typicode.com/posts/1',
function(responseData) {
  const data = JSON.parse(responseData);
  console.log(data);
}, function(error) {
  console.error(error);
});
```

In this example, we're passing two callback functions as arguments to the makeHttpRequest function. The successCallback function will be called if the HTTP request is successful, and the errorCallback function will be called if an error occurs. We're also handling the error event separately and calling the errorCallback function with an appropriate error message.

UNDERSTANDING PROMISES

Promises are a newer method of asynchronous programming that was introduced in ECMAScript 6. Promises are a way of handling asynchronous operations in JavaScript. A promise is an object that represents a future value, which may or may not be available yet. Promises provide a cleaner and more flexible way to handle asynchronous operations compared to traditional callback functions. A promise can have three states:

- **Pending**: The initial state, the promise is neither fulfilled nor rejected yet.
- **Fulfilled**: The promise has been completed, and the result value is available.

- **Rejected**: The promise has failed, and the error message is available.

Here's an example of a promise that represents a simulated HTTP request:

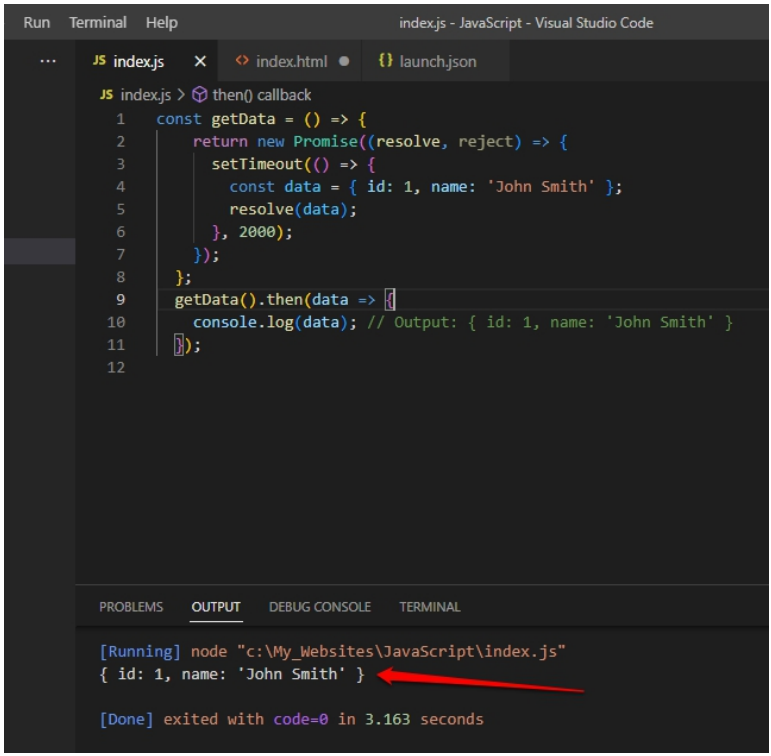
```
const getData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = { id: 1, name: 'John Smith ' };
      resolve(data);
    }, 2000);
  });
};
```

In this example, getData() returns a new promise. Inside the promise, there is a setTimeout() function that simulates a delay of 2 seconds. Once the delay is complete, the resolve() function is called with a sample data object as an argument.

To use this promise, you can call the then() method on it, passing in a callback function that will be executed once the promise is fulfilled:

```
getData().then(data => {
  console.log(data); // Output: { id: 1, name: 'John Smith' }
});
```

In this example, the then() method takes a callback function that takes the resolved data as an argument. The console.log() statement will output the data object to the console. See the output below.

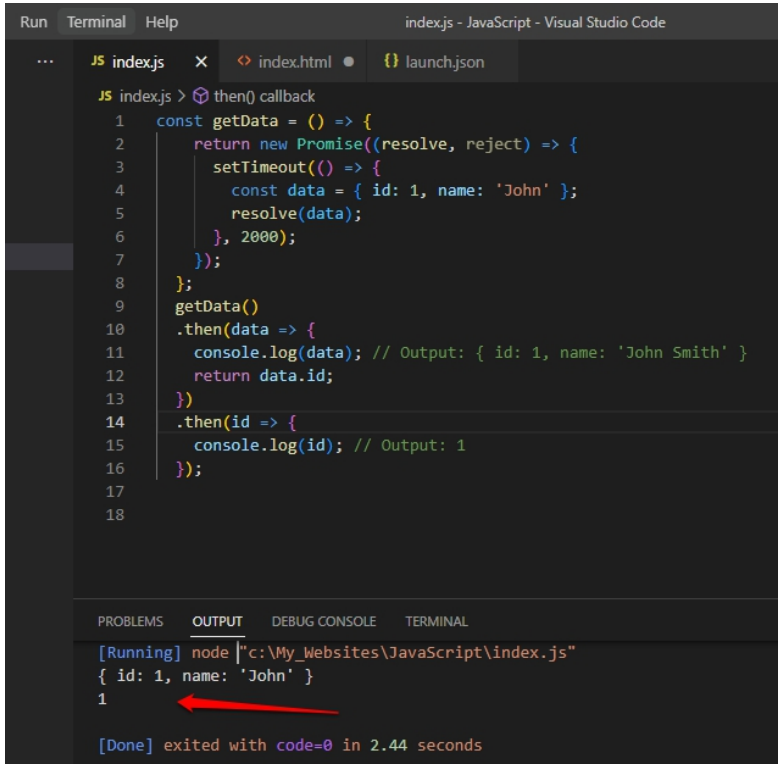


You can also chain multiple then() methods to handle the resolved data:

```
getData()
  .then(data => {
    console.log(data); // Output: { id: 1, name: 'John Smith' }
    return data.id;
  })
  .then(id => {
    console.log(id); // Output: 1
  });
```

In this example, the first then() method returns the id property of the data object, which is passed as an argument to the second then() method. The console.log() statement inside the second then()

method outputs the id to the console. See the output below.

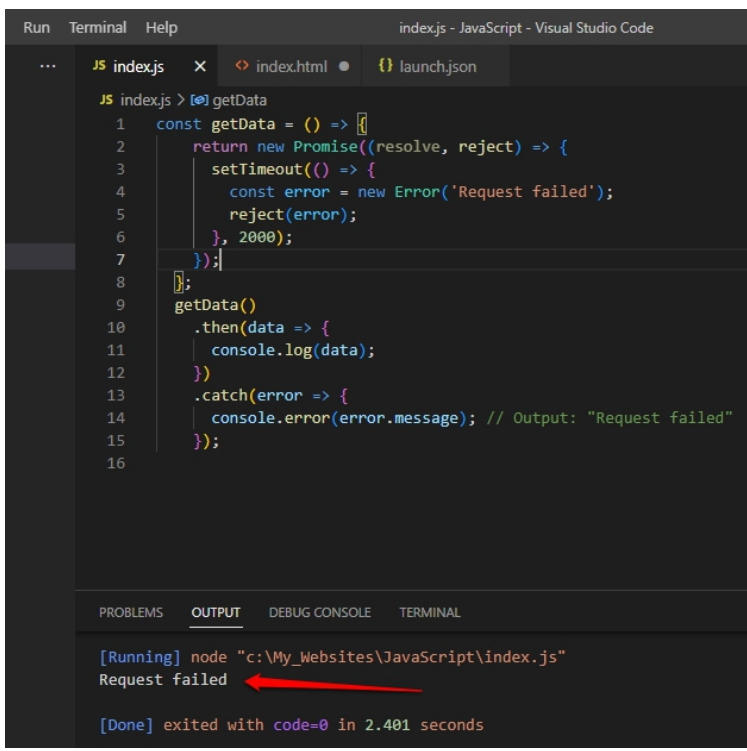


You can also handle errors with promises using the catch() method:

```
const getData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const error = new Error('Request failed');
      reject(error);
    }, 2000);
  });
};
getData()
  .then(data => {
```

```
  console.log(data);
})
.catch(error => {
  console.error(error.message); // Output: "Request failed"
});
```

In this example, the promise is intentionally rejected with an error message. The catch() method is used to handle the error and output the error message to the console. See the output below.



Promises provide a cleaner and more flexible way to handle asynchronous operations compared to traditional callback functions. They allow you to write more readable and maintainable code, with better error handling and chaining of operations.

ASYNC/AWAIT

Async/await is a newer way to handle asynchronous code in

JavaScript, introduced in ES2017. It is built on top of Promises and provides a more readable and cleaner syntax for handling asynchronous operations. The async keyword is used to declare an asynchronous function, which automatically returns a Promise. Within an asynchronous function, the await keyword is used to wait for the completion of a Promise before continuing with the next line of code.

Here's an example of an asynchronous function that fetches data from an API using the fetch method and returns the result using a Promise:

```
async function getData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
}
```

In the above code, the await keyword is used to wait for the completion of the fetch method, which returns a Promise that resolves to a Response object. The JSON method of the Response object also returns a Promise, which is resolved to the actual JSON data. By using the await keyword, we wait for both Promises to resolve before returning the data.

We can also use try/catch blocks with async/await functions to handle errors. Here's an example:

```
async function getData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
  }
```

```
}
```

In the above code, if an error occurs during the fetching or parsing of the data, the error will be caught in the catch block and logged to the console.

Async/await can also be used with Promise.all to fetch multiple resources in parallel. Here's an example:

```
async function getData() {
  const [userData, postData] = await Promise.all([
    fetch('https://api.example.com/users'),
    fetch('https://api.example.com/posts')
  ]);
  const users = await userData.json();
  const posts = await postData.json();
  return { users, posts };
}
```

In the above code, we use Promise.all to fetch the user and post data simultaneously. The await keyword is then used to wait for both Promises to resolve before parsing the JSON data and returning it as an object.

CHAPTER TWENTY

AJAX AND APIS

This chapter focuses on AJAX (Asynchronous JavaScript and XML) and APIs (Application Programming Interfaces), which are essential tools for building dynamic and interactive web applications. AJAX is a technique for exchanging data with a server, updating parts of a web page without refreshing the entire page, and providing a seamless user experience. APIs, on the other hand, provide a way for different software applications to communicate with each other, enabling developers to access data and functionality from third-party services and integrate them into their applications.

In this chapter, we will cover the basics of AJAX and APIs, including how to work with APIs, how to fetch data using AJAX, and how to display the data on a webpage. We will also discuss some common API integration scenarios, such as social media integration, weather data integration, and more. By the end of this chapter, you will have a solid understanding of how to use AJAX and APIs to create powerful, dynamic, and interactive web applications.

INTRODUCTION TO AJAX

AJAX stands for Asynchronous JavaScript and XML. It is a technique used in web development to create dynamic web pages that can exchange data with a web server without having to reload the entire page. AJAX is based on JavaScript and uses a combination of XMLHttpRequest (XHR) objects and the Document Object Model (DOM) to interact with the server and update the web page asynchronously.

Before AJAX, web pages had to be reloaded entirely whenever new data was requested, resulting in a slow and inefficient user experience. With AJAX, web pages can update a portion of the content without having to reload the entire page, making the user experience much smoother and faster.

AJAX is widely used in web development for a variety of purposes, such as real-time chat applications, dynamic web forms, and auto-suggest search bars. It also allows developers to create web

applications that can interact with APIs to fetch and display data in real-time.

WORKING WITH APIS

API stands for "Application Programming Interface." It is a set of protocols, routines, and tools for building software applications. APIs provide a way for different software components to interact with each other. In web development, an API is often used to communicate with a server and retrieve data or perform other operations.

To work with an API, you need to know the endpoint or URL of the API, along with any required authentication or API keys. You also need to know the format of the data that the API returns, such as JSON or XML.

Here are some examples of working with APIs in JavaScript:

Fetching data from an API using the Fetch API

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, we use the Fetch API to make a GET request to an API endpoint at https://api.example.com/data. The response is then converted to JSON using the response.json() method. The resulting data is logged into the console.

Sending data to an API using the Fetch API

```
fetch('https://api.example.com/data', {
  method: 'POST',
  body: JSON.stringify({ name: 'John Doe', email: 'johndoe@example.com' }),
  headers: { 'Content-Type': 'application/json' }
})
```

```
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, we use the Fetch API to make a POST request to an API endpoint at https://api.example.com/data. We include data in the request body as a JSON object containing a name and email property. We also set the Content-Type header to application/json. The response is then converted to JSON using the response.json() method. The resulting data is logged into the console.

Using a third-party API wrapper library like Axios

```
axios.get('https://api.example.com/data')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

In this example, we use the Axios library to make a GET request to an API endpoint at https://api.example.com/data. The response data is logged into the console.

Implementing APIs in a Practical Exercise

In this exercise, we will build a simple weather app that fetches weather data from an API and displays it on a webpage. The user can enter a location and the app will display the current weather conditions for that location. We will be using the OpenWeatherMap API to fetch weather data for our app. The API provides a free plan that allows us to make up to 60 requests per minute.

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Weather App</title>
    <link rel="stylesheet" href="index.css">
  </head>
  <body>
    <h1>Weather App</h1>
    <form>
      <label for="location1">Enter location:</label>
      <input type="text" id="location1" name="location">
      <button type="button" class="get-weather">Get Weather</button>
    </form>
    <div id="weather1"></div>
    <form>
      <label for="location2">Enter location:</label>
      <input type="text" id="location2" name="location">
      <button type="button" class="get-weather">Get Weather</button>
    </form>
    <div id="weather2"></div>
    <script src="index.js"></script>
</body>
</html>
```

CSS Code:

```
/* Set global styles */
body {
```

```
  font-family: Arial, sans-serif;
  background-color: #f8f8f8;
}
  h1 {
  text-align: center;
  margin-top: 30px;
}
  form {
  display: flex;
  flex-direction: row;
  align-items: center;
  margin: 20px 0;
}
  label {
  margin-right: 10px;
}
  input[type="text"] {
  padding: 5px;
  border-radius: 5px;
  border: none;
  margin-right: 10px;
}
  button.get-weather {
  padding: 5px 10px;
  border-radius: 5px;
  border: none;
```

```
    background-color: #4CAF50;
    color: #fff;
    cursor: pointer;
  }
    button.get-weather:hover {
    background-color: #3e8e41;
  }
    #weather1,
  #weather2 {
    margin: 20px 0;
    padding: 10px;
    border-radius: 5px;
    border: 1px solid #ccc;
    background-color: #fff;
  }
```

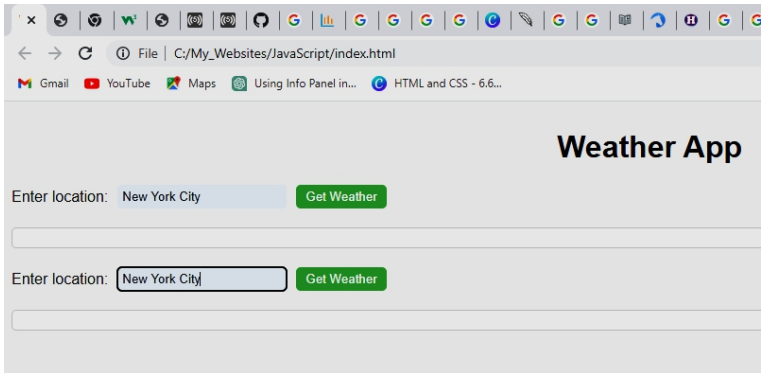
JavaScript Code:

```
// Get all the form elements
const forms = document.querySelectorAll('form');
// Loop through each form
forms.forEach((form, index) => {
  const locationInput = form.querySelector('input[type="text"]');
  const weatherDiv = document.querySelector(`#weather${index + 1}
`);
  const button = form.querySelector('button');
  // Add event listener to the button
  button.addEventListener('click', () => {
```

```
    const location = locationInput.value;
    const apiKey = 'YOUR_API_KEY_HERE';
    const apiUrl = `https://api.openweathermap.org/data/2.5/weather?q=${location}&appid=${apiKey}&units=metric`;
    // Fetch weather data
    fetch(apiUrl)
      .then(response => response.json())
      .then(data => {
        const temperature = data.main.temp;
        const description = data.weather[0].description;
        const city = data.name;
        const country = data.sys.country;
        // Update the weather data
        weatherDiv.innerHTML = `
          <p><strong>Location:</strong> ${city}, ${country}</p>
          <p><strong>Temperature:</strong> ${temperature} &deg;C</p>
          <p><strong>Condition:</strong> ${description}</p>
        `;
      })
      .catch(error => {
        console.error(error);
        weatherDiv.textContent = 'An error occurred while fetching the weather data.';
      });
  });
```

});

See the output below.



FETCHING DATA WITH AJAX

Fetching data with AJAX allows us to make requests to APIs or servers without having to reload the entire page. This makes web applications more dynamic and responsive to user actions.

AJAX stands for "Asynchronous JavaScript and XML", but nowadays JSON is often used instead of XML. In JavaScript, we can use the XMLHttpRequest (XHR) object to send and receive data from a server asynchronously. However, the Fetch API is a newer and more modern way of doing this in JavaScript.

Here's an example of fetching data with AJAX using the Fetch API:

```
const apiUrl = 'https://api.example.com/data';

fetch(apiUrl)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
```

```
  console.log(data);
})
.catch(error => {
  console.error('There was a problem fetching the data:', error);
});
```

In this example, we use the fetch() method to make a GET request to the URL specified in the apiUrl variable. The fetch() method returns a Promise that resolves to the Response object representing the server's response to the request. We can then check if the response was successful using the ok property of the Response object. If it was not successful, we can throw an Error with a custom message. If the response was successful, we can extract the JSON data using the json() method of the Response object, which also returns a Promise.

Finally, we can use another .then() block to log the retrieved data to the console, or do something else with it. If there was an error at any point, we can use the .catch() block to handle it and log a custom error message to the console.

Note that the Fetch API is supported in modern web browsers, but not in older versions of Internet Explorer. To support older browsers, we can use a polyfill or a library like jQuery to make AJAX requests.

DISPLAYING DATA ON A WEBPAGE

Once you have successfully fetched data using AJAX, the next step is to display it on your webpage. You can display the data in various ways, such as updating an HTML element, creating new elements dynamically, or generating charts.

Here's an example of how you can update an HTML element with the fetched data:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Displaying Data</title>
    <link rel="stylesheet" href="index.css">
  </head>
  <body>
    <div id="data-container">
      <h1>Loading data...</h1>
    </div>
    <script src="index.js"></script>
</body>
</html>
```

CSS Code:

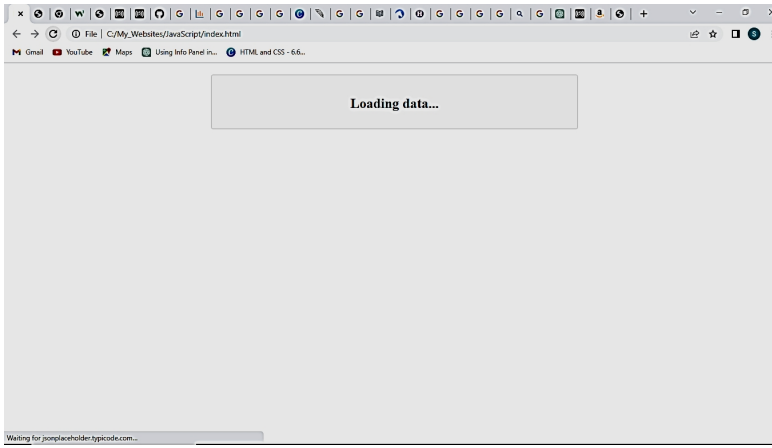
```
#data-container {
    margin: 20px auto;
    padding: 20px;
    max-width: 600px;
    text-align: center;
    background-color: #f5f5f5;
    border: 1px solid #ddd;
    border-radius: 4px;
  }
  h1 {
```

```
    font-size: 24px;
    font-weight: bold;
    margin-bottom: 10px;
  }
```

JavaScript Code:

```
const dataContainer = document.querySelector('#data-container');
const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';
fetch(apiUrl)
  .then(response => response.json())
  .then(data => {
    const title = data.title;
    const body = data.body;
    const html = `
      <h1>${title}</h1>
      <p>${body}</p>
    `;
    dataContainer.innerHTML = html;
  })
  .catch(error => console.error(error));
```

See the output below.



CHAPTER TWENTY ONE

UNDERSTANDING ES6

ES6, also known as ECMAScript 2015, is a major update to the JavaScript language that introduced many new features and syntax improvements. These new features provide developers with more powerful and efficient ways to write JavaScript code.

In this chapter, we'll explore some of the most important new features introduced in ES6, including arrow functions, template literals, destructuring, and classes. We'll also cover how to work with modules to better organize and reuse our code.

we'll also look ahead to some of the upcoming features and updates

to the JavaScript language that is currently being developed and will be introduced in future versions of ECMAScript.

INTRODUCTION TO ES6

ECMAScript 6 (ES6), also known as ECMAScript 2015, is a major update to the JavaScript language that was released in June 2015. ES6 introduced many new features and syntax to JavaScript that make it easier to write and read code. It also added new built-in objects and methods that make common programming tasks simpler and more efficient. ES6 introduced several significant changes to the language, including:

- Block-scoped variables (let and const)
- Arrow functions
- Template literals
- Destructuring
- Classes and modules
- Default parameters
- Rest and spread operators
- Promises
- Generators
- Symbols

NEW FEATURES AND SYNTAX

ES6 introduced some new features and syntax to JavaScript. Some of the most notable ones are:

- **Let and const:** ES6 introduced two new keywords for declaring variables: let and const. Unlike the var keyword, which has function scope, let and const have block scope. Variables declared with let can be reassigned, while variables declared with const are read-only.
- **Arrow functions:** Arrow functions provide a more concise syntax for defining functions. They are anonymous functions that use a short arrow notation (=>) instead of the function keyword. Arrow functions can also be used as shorthand for returning a value from a function.

- **Template literals:** Template literals provide a way to embed expressions inside string literals. They use backticks (`) instead of single or double quotes, and allow you to include variables and expressions inside ${}.
- **Destructuring:** Destructuring provides a way to extract values from arrays and objects and assign them to variables. It uses a shorthand syntax to make it easier to extract multiple values at once.
- **Classes and modules:** ES6 introduced a class syntax that provides a more traditional object-oriented programming model. Classes allow you to define a blueprint for creating objects with methods and properties. ES6 also introduced a module syntax that makes it easier to organize and share code between files.
- **Default parameters:** Default parameters provide a way to specify a default value for a function parameter if no value is provided.
- **Rest and spread operators:** The rest and spread operators provide a way to work with arrays and objects more easily. The rest operator (...) allows you to collect multiple arguments into an array, while the spread operator (...) allows you to spread an array or object into individual arguments.
- **Promises:** Promises provide a way to handle asynchronous operations in a more readable and predictable way. They allow you to chain together asynchronous operations and handle errors more gracefully.
- **Generators:** Generators provide a way to pause and resume the execution of a function. They are useful for creating iterators that can be used to iterate over a sequence of values.
- **Symbols:** Symbols provide a way to create unique, immutable values that can be used as keys in objects. They are useful for creating private properties and methods.

WORKING WITH MODULES

In ES6, the concept of modules was introduced to help organize and reuse code more efficiently. A module is simply a separate file containing code that can be imported and used in another file.

The export keyword is used in a module file to make certain parts of the code available for use in other files. For example, if we have a file named utils.js with a function sum that we want to use in another file, we can export the sum function like this:

```
// utils.js

export function sum(a, b) {

  return a + b;

}
```

In another file, we can import the sum function using the import keyword like this:

```
// app.js

import { sum } from './utils.js';

console.log(sum(2, 3)); // output: 5
```

The import statement above imports the sum function from the utils.js module and makes it available for use in the app.js file.

Another way to export code from a module is to use the export default syntax. With this syntax, we can export a single value from a module without needing to use the braces when importing it. For example:

```
// utils.js

export default function sum(a, b) {

  return a + b;

}
```

In another file, we can import the default export like this:

```
// app.js

import sum from './utils.js';

console.log(sum(2, 3)); // output: 5
```

Note that when using export default, we don't need to use the braces when importing the exported value. We can simply use the name we want to assign to the imported value (in this case, sum).

FUTURE UPDATES TO JAVASCRIPT

JavaScript is a constantly evolving language, with new updates and proposals being added regularly. Some of the upcoming features and updates to JavaScript include:

- **Optional Chaining:** This feature allows developers to safely access nested properties of an object without worrying about whether each property exists. This helps reduce the amount of boilerplate code needed for null checks and improves code readability.
- **Nullish Coalescing Operator:** This operator allows developers to check for null or undefined values and concisely provide a default value. It helps reduce the amount of boilerplate code needed for null checks and provides more readable code.
- **Promise.any():** This feature allows developers to create a Promise that resolves when any of the promises passed to it resolve. This is useful when you want to perform multiple async operations and get a result as soon as any of them complete successfully.
- **Logical Assignment Operators:** These operators are shorthand for common operations, such as `+=`, `-=`, `*=`, and `/=`, combined with logical operators. For example, the `||=` operator assigns a value to a variable only if it is null or undefined.
- **Private Fields and Methods**: This feature provides a way to create private properties and methods in classes, making it easier to encapsulate and protect data.
- **SIMD (Single Instruction, Multiple Data):** This feature enables JavaScript to perform SIMD operations on arrays, making it faster and more efficient to process large amounts of data.
- **GlobalThis:** This feature provides a way to access the global object in a cross-platform way, regardless of the environment in which the code is running.

These are just a few examples of the many updates and proposals for JavaScript. As the language continues to evolve, developers need to stay up to date with the latest features and syntax.

COMMON MISTAKES TO AVOID WHEN WRITING JAVASCRIPT CODE

It is important to follow best practices to ensure that your code is clean, maintainable, and easy to read. In this article, we will discuss some common mistakes to avoid when writing JavaScript code and some best practices for writing clean, maintainable code.

Common Mistakes to Avoid

- **Not using const and let**: One of the most common mistakes is to use var instead of const or let when declaring variables. The var keyword can lead to problems when used in loops or nested functions, as it is not block-scoped. In contrast, const and let are block-scoped and are a better choice for declaring variables.
- **Not checking for undefined**: Another common mistake is not checking for undefined values. This can lead to errors when trying to access properties of undefined objects or calling undefined functions. To avoid this, always check for undefined values before accessing them.
- **Using global variables**: Global variables can cause problems when used in large applications, as they can be accessed and modified from any part of the application. Instead, use local variables or encapsulate your code in modules to avoid using global variables.
- **Not using strict mode**: Strict mode is a feature in JavaScript that helps catch common coding mistakes and makes it easier to write secure code. Always use strict mode in your JavaScript code to avoid common mistakes.
- **Not handling errors**: Error handling is an important aspect of writing JavaScript code. Not handling errors can lead to unexpected behavior and can make it difficult to debug code. Always handle errors by using

try-catch blocks or by returning error objects.

BEST PRACTICES FOR WRITING CLEAN, MAINTAINABLE CODE

- **Use descriptive names**: Use descriptive names for variables, functions, and classes to make it easier to understand what they do. Avoid using single-letter or generic names, as they can make it difficult to understand the purpose of the code.
- **Write modular code**: Modular code is code that is organized into smaller, reusable modules. This makes it easier to maintain and debug code and reduces the likelihood of introducing bugs. Use modules to encapsulate your code and avoid using global variables.
- **Use comments:** Use comments to explain the purpose of the code, how it works, and any potential issues or gotchas. This makes it easier for other developers to understand and maintain the code.
- **Avoid nested code:** Avoid using nested loops or nested functions, as they can make the code difficult to read and understand. Instead, use modular code and functions to break down the code into smaller, more manageable pieces.
- **Use proper indentation:** Proper indentation is important for making code easier to read and understand. Use consistent indentation for each level of code, and avoid mixing tabs and spaces.
- **Use semicolons**: Although not required in JavaScript, it is a good practice to use semicolons to terminate statements. This makes the code easier to read and avoids potential errors.
- **Use const and let**: Use const and let instead of var when declaring variables. This makes the code easier to read and understand and avoids potential issues with scoping.
- **Use template literals:** Use template literals instead of concatenation when building strings. Template literals make it easier to build complex strings and are more readable than concatenation.

- **Use arrow functions:** Use arrow functions instead of traditional function declarations. Arrow functions are more concise and easier to read.
- **Use destructuring:** Use destructuring to extract values from objects and arrays. Destructuring makes it easier to work with complex data structures and is more readable than traditional methods.

CHAPTER TWENTY-TWO

PRACTICAL EXERCISES

JavaScript Practical Exercises are a great way for developers to practice their coding skills by building real-world applications. These exercises provide an opportunity to apply the concepts learned in JavaScript courses and tutorials to practical scenarios. Projects can vary in complexity and scope, ranging from simple to-do list apps to full-fledged web applications with backend servers.

Working on project exercises can help developers gain confidence in their coding abilities, learn new techniques, and develop problem-solving skills. It also provides an opportunity to explore new frameworks and libraries and gain practical experience in working with them.

Moreover, JavaScript Practical Exercises can be a great addition to a developer's portfolio. Potential employers can evaluate a developer's skills by reviewing their completed projects. Additionally, it can also help developers to network and collaborate with other developers working on similar projects.

In this chapter, we will explore some project ideas that developers can work on to improve their JavaScript skills. Each project will

have a brief description, and we will guide the key features and functionalities to be implemented. We will also provide code snippets and helpful resources to aid in the development process.

Note that the code listing in this book is only meant to guide you in creating your desired website or web application, you can add or remove any element of your choice.

EXERCISE 1: CREATING A SIMPLE APP TO MANAGE TASKS AND TO-DO LISTS

Below is an example of a simple to-do list app written in JavaScript and its equivalent HTML and CSS code:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-do List</title>
    <link rel="stylesheet" href="index.css">
</head>
<body>
  <h1>To-do List</h1>
  <input type="text" id="new-todo" placeholder="Enter new to-do item">
  <button id="add-todo">Add</button>
  <ul id="todo-list"></ul>
    <script src="index.js"></script>
```

```
</body>
</html>
```

CSS Code:

```
ul {
    list-style: none;
    padding-left: 0;
  }
  li {
    margin-bottom: 5px;
  }
  .completed {
    text-decoration: line-through;
  }
```

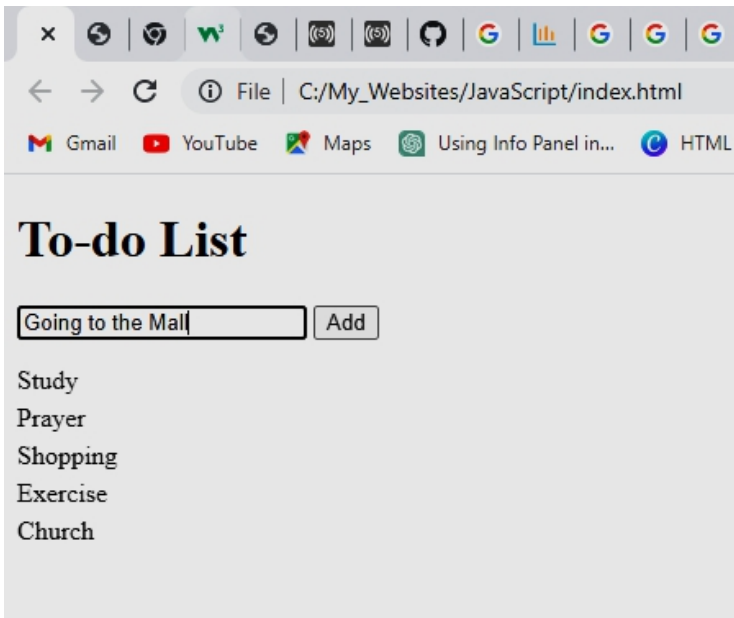
JavaScript Code:

```
const todoList = document.getElementById('todo-list');
const newTodoInput = document.getElementById('new-todo');
const addTodoButton = document.getElementById('add-todo');
let todos = [];
// Render the list of todos
function renderTodos() {
  todoList.innerHTML = '';
  todos.forEach(function(todo, index) {
    const li = document.createElement('li');
    li.textContent = todo.text;
    if (todo.completed) {
      li.classList.add('completed');
```

```
    }
    li.addEventListener('click', function() {
      toggleCompleted(index);
    });
    todoList.appendChild(li);
  });
}
// Add a new todo
function addTodo() {
  const newTodoText = newTodoInput.value.trim();
  if (newTodoText === '') {
    return;
  }
  todos.push({
    text: newTodoText,
    completed: false
  });
  newTodoInput.value = '';
  renderTodos();
}
// Toggle the completed state of a todo
function toggleCompleted(index) {
  todos[index].completed = !todos[index].completed;
  renderTodos();
}
// Add event listeners
```

```
addTodoButton.addEventListener('click', addTodo);
newTodoInput.addEventListener('keydown', function(event) {
  if (event.key === 'Enter') {
    addTodo();
  }
});
// Initial render
renderTodos();
```

See the screenshot below.



EXERCISE 2: CREATING A CALCULATOR APP TO PERFORM SIMPLE AND COMPLEX MATHEMATICAL OPERATIONS

an example of a basic calculator app that can perform simple and complex mathematical operations using HTML, CSS, and JavaScript:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Calculator</title>
    <link rel="stylesheet" href="index.css">
</head>
<body>
    <div class="calculator">
        <div class="output">
            <input type="text" id="result" readonly>
        </div>
        <div class="buttons">
            <button onclick="clearResult()">C</button>
            <button onclick="backspace()">&larr;</button>
            <button onclick="calculate('^')">x<sup>y</sup></button>
            <button onclick="calculate('/')">&divide;</button>
            <button onclick="calculate('*')">&times;</button>
            <button onclick="calculate('-')">-</button>
            <button onclick="calculate('+')">+</button>
            <button onclick="calculate('%')">%</button>
            <button onclick="calculate('.')">.</button>
            <button onclick="calculate('0')">0</button>
```

```
<button onclick="calculate('1')">1</button>
<button onclick="calculate('2')">2</button>
<button onclick="calculate('3')">3</button>
<button onclick="calculate('4')">4</button>
<button onclick="calculate('5')">5</button>
<button onclick="calculate('6')">6</button>
<button onclick="calculate('7')">7</button>
<button onclick="calculate('8')">8</button>
<button onclick="calculate('9')">9</button>
<button onclick="calculate('(')">(</button>
<button onclick="calculate(')')">)</button>
<button onclick="calculate('pi')">&pi;</button>
<button onclick="calculate('e')">e</button>
<button onclick="calculate('sqrt(')">√</button>
<button onclick="calculate('sin(')">sin</button>
<button onclick="calculate('cos(')">cos</button>
<button onclick="calculate('tan(')">tan</button>
<button onclick="calculate('log(')">log</button>
<button onclick="calculate('ln(')">ln</button>
<button onclick="calculate('abs(')">|x|</button>
<button onclick="calculate('floor(')">floor</button>
<button onclick="calculate('ceil(')">ceil</button>
<button onclick="calculate('exp(')">exp</button>
<button onclick="calculate('min(')">min</button>
<button onclick="calculate('max(')">max</button>
<button onclick="calculate('rand')">rand</button>
```

```
            <button onclick="equal()">=</button>
        </div>
    </div>
    <script src="index.js"></script>
</body>
</html>
```

CSS Code:

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f1f1f1;
  }
    .calculator {
    max-width: 500px;
    margin: 0 auto;
    background-color: #fff;
    border: 1px solid #ccc;
    border-radius: 5px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.3);
    padding: 20px;
    box-sizing: border-box;
  }
    .output {
    margin-bottom: 10px;
  }
```

```
  .output input {
  width: 100%;
  padding: 10px;
  font-size: 24px;
  border: 1px solid #ccc;
  border-radius: 5px;
  box-sizing: border-box;
}
  .buttons {
  display: grid;
  grid-template-columns: repeat(5, 1fr);
  grid-gap: 10px;
}
  button {
  padding: 10px;
  font-size: 20px;
  background-color: #4caf50;
  color: #fff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  box-shadow: 0 2px 5px rgba(0, 0, 0, 0.3);
  transition: all 0.2s ease-in-out;
}
  button:hover {
  background-color: #3e8e41;
```

```
  }
```

JavaScript Code:

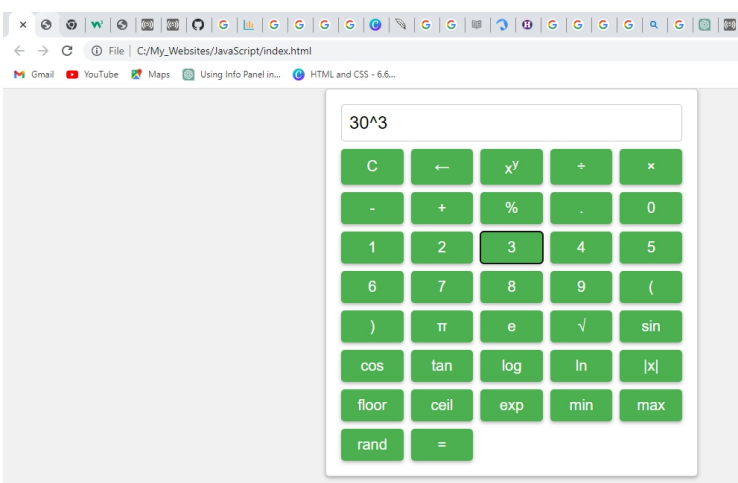
```
let result = document.getElementById("result");

function calculate(input) {
  if (input === "C") {
    result.value = "";
  } else if (input === "rand") {
    result.value += Math.random().toFixed(2);
  } else if (input === "pi") {
    result.value += Math.PI.toFixed(2);
  } else if (input === "e") {
    result.value += Math.E.toFixed(2);
  } else if (input === "sqrt(") {
    result.value += "sqrt(";
  } else if (input === "sin(") {
    result.value += "sin(";
  } else if (input === "cos(") {
    result.value += "cos(";
  } else if (input === "tan(") {
    result.value += "tan(";
  } else if (input === "log(") {
    result.value += "log(";
  } else if (input === "ln(") {
    result.value += "ln(";
  } else if (input === "abs(") {
```

```
    result.value += "abs(";
  } else if (input === "floor(") {
    result.value += "floor(";
  } else if (input === "ceil(") {
    result.value += "ceil(";
  } else if (input === "exp(") {
    result.value += "exp(";
  } else if (input === "min(") {
    result.value += "min(";
  } else if (input === "max(") {
    result.value += "max(";
  } else {
    result.value += input;
  }
}
function backspace() {
  result.value = result.value.substring(0, result.value.length - 1);
}
function equal() {
  let expression = result.value;
  if (expression.includes("^")) {
    expression = expression.replace("^", "**");
  }
  if (expression.includes("sqrt(")) {
    expression = expression.replace("sqrt(", "Math.sqrt(");
  }
```

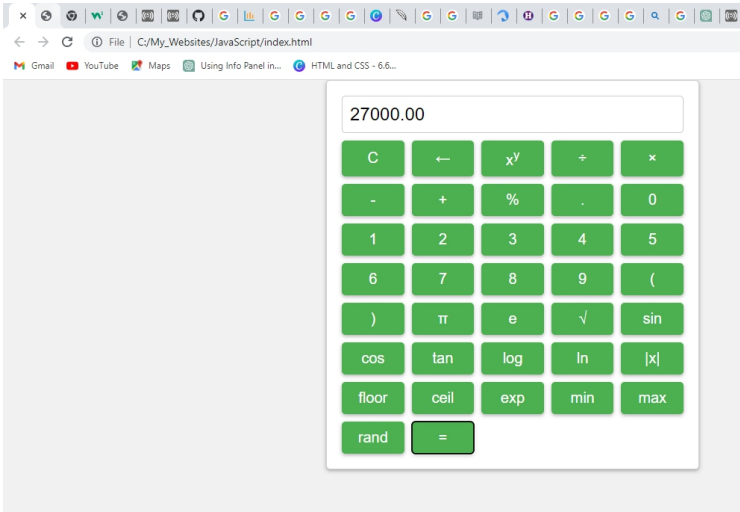
```
if (expression.includes("sin(")) {
  expression = expression.replace("sin(", "Math.sin(");
}
if (expression.includes("cos(")) {
  expression = expression.replace("cos(", "Math.cos(");
}
if (expression.includes("tan(")) {
  expression = expression.replace("tan(", "Math.tan(");
}
if (expression.includes("log(")) {
  expression = expression.replace("log(", "Math.log10(");
}
if (expression.includes("ln(")) {
  expression = expression.replace("ln(", "Math.log(");
}
if (expression.includes("abs(")) {
  expression = expression.replace("abs(", "Math.abs(");
}
if (expression.includes("floor(")) {
  expression = expression.replace("floor(", "Math.floor(");
}
if (expression.includes("ceil(")) {
  expression = expression.replace("ceil(", "Math.ceil(");
}
if (expression.includes("exp(")) {
  expression = expression.replace("exp(", "Math.exp(");
```

```
  }
  if (expression.includes("min(")) {
    expression = expression.replace("min(", "Math.min(");
  }
  if (expression.includes("max(")) {
    expression = expression.replace("max(", "Math.max(");
  }
  try {
    result.value = eval(expression).toFixed(2);
  } catch (error) {
    result.value = "Error";
  }
}
function clearResult() {
  result.value = "";
}
```

See the output below.



See the result of the calculation below.



EXERCISE 3: CREATING A SIMPLE GAME USING JAVASCRIPT

Below is an example of a simple game called "Whack-a-Mole" created using HTML, CSS, and JavaScript:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Whack-a-Mole Game</title>
    <link rel="stylesheet" type="text/css" href="index.css">
</head>
<body>
```

```
    <div class="game">
        <h1>Whack-a-Mole</h1>
        <p>Score: <span id="score">0</span></p>
        <div class="holes">
            <div class="hole hole1">
                <div class="mole"></div>
            </div>
            <div class="hole hole2">
                <div class="mole"></div>
            </div>
            <div class="hole hole3">
                <div class="mole"></div>
            </div>
            <div class="hole hole4">
                <div class="mole"></div>
            </div>
            <div class="hole hole5">
                <div class="mole"></div>
            </div>
            <div class="hole hole6">
                <div class="mole"></div>
            </div>
        </div>
    </div>
    <script type="text/javascript" src="index.js"></script>
</body>
```

</html>

CSS Code:

```
.game {
    text-align: center;
    margin-top: 100px;
}
.holes {
    display: flex;
    justify-content: center;
    align-items: center;
    margin-top: 50px;
}
.hole {
    display: inline-block;
    width: 100px;
    height: 100px;
    background-color: #C2B280;
    margin: 10px;
    position: relative;
}
.mole {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
```

```
    background-image: url('https://image.flaticon.com/icons/svg/174/174879.svg');
    background-repeat: no-repeat;
    background-position: center;
    background-size: 60%;
    transition: all 0.3s;
}
.mole.up {
    top: -60px;
}
.mole.hit {
    background-image: url('https://image.flaticon.com/icons/svg/174/174875.svg');
}
p {
    font-size: 24px;
    margin-bottom: 20px;
}
#score {
    font-weight: bold;
    color: #FF6F00;
}
```

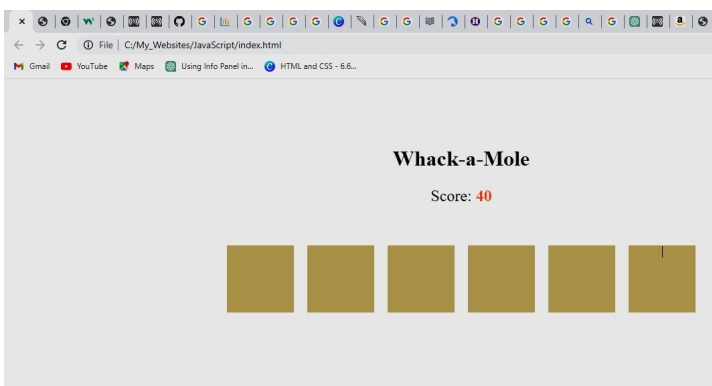
JavaScript Code:

```
const holes = document.querySelectorAll('.hole');
const scoreBoard = document.querySelector('#score');
const moles = document.querySelectorAll('.mole');
let score = 0;
```

```
let lastHole;
let timeUp = false;
function randomTime(min, max) {
  return Math.round(Math.random() * (max - min) + min);
}
function randomHole(holes) {
  const idx = Math.floor(Math.random() * holes.length);
  const hole = holes[idx];
  if (hole === lastHole) {
    return randomHole(holes);
  }
  lastHole = hole;
  return hole;
}
function peep() {
  const time = randomTime(200, 1000);
  const hole = randomHole(holes);
  hole.classList.add('up');
  setTimeout(() => {
    hole.classList.remove('up');
    if (!timeUp) {
      peep();
    }
  }, time);
}
function bonk(e) {
```

```
  if (!e.isTrusted) {
    return;
  }
  score++;
  this.classList.remove('up');
  scoreBoard.textContent = score;
}
moles.forEach((mole) => mole.addEventListener('click', bonk));
function startGame() {
  scoreBoard.textContent = 0;
  score = 0;
  timeUp = false;
  peep();
  setTimeout(() => timeUp = true, 10000);
}
```

See the screenshot below.



DEPLOYMENT AND HOSTING OPTIONS.

Deployment and hosting are critical aspects of software development that involve making applications available to users on

various devices and platforms. There are several deployment and hosting options, each with its pros and cons, depending on the type of application, budget, and desired level of control. In this response, we will discuss some of the most common deployment and hosting options available.

- **On-premises deployment**: This deployment option involves running applications on servers or other hardware located on-site. On-premises deployment gives businesses greater control over their applications and data, but it can also be expensive to maintain and upgrade the hardware, and it requires an experienced IT team to manage the infrastructure.
- **Cloud deployment**: Cloud deployment involves running applications on remote servers hosted by cloud service providers like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Cloud deployment is scalable, flexible, and cost-effective since businesses pay only for the resources they use. Cloud service providers also offer various services like databases, storage, and analytics, among others, making it easy for businesses to integrate different services and features into their applications.
- **Virtual Private Server (VPS) hosting**: VPS hosting involves renting a virtual server from a hosting provider. VPS hosting offers more control over the server environment than shared hosting, and it is also more affordable than dedicated hosting. However, VPS hosting requires some technical knowledge to set up and manage, and businesses need to ensure that the server resources are enough to meet their application's needs.
- **Shared hosting**: Shared hosting involves hosting applications on a server shared with other users. Shared hosting is the most affordable hosting option, making it ideal for small businesses or individuals with small applications. However, shared hosting offers limited control over the server environment, and the shared resources may cause performance issues.
- **Dedicated hosting**: Dedicated hosting involves renting

a physical server dedicated solely to an application. Dedicated hosting offers complete control over the server environment, and it is ideal for applications with high traffic or resource-intensive applications. However, dedicated hosting is expensive, and it requires technical knowledge to set up and manage.

- **Containerization**: Containerization involves packaging applications into containers that can run consistently across different platforms and environments. Containerization offers a standardized environment where developers can build, test, and deploy applications. Containerization also allows for easy scaling, updating, and maintaining of applications. Containerization technologies include Docker and Kubernetes.

UNDERSTANDING SEVERAL CHEAPER PLATFORMS TO HOST YOUR WEBSITES

Several web hosting providers offer affordable hosting plans for individuals and small businesses. Here are some of the cheapest options:

- **Hostinger**: Hostinger offers shared hosting plans starting at just $0.99 per month. Their plans come with a free domain, website builder, and 24/7 customer support.
- **Bluehost**: Bluehost is a popular web hosting provider that offers shared hosting plans starting at $2.95 per month. Their plans include a free domain, SSL certificate, and website builder.
- **DreamHost**: DreamHost offers shared hosting plans starting at $2.59 per month. Their plans come with unlimited storage and bandwidth, a free domain, and 24/7 customer support.
- **A2 Hosting**: A2 Hosting offers shared hosting plans starting at $2.99 per month. Their plans include unlimited storage and bandwidth, a free SSL certificate, and 24/7 customer support.
- **HostGator**: HostGator is another popular web hosting

provider that offers shared hosting plans starting at $2.75 per month. Their plans include a free domain, website builder, and 24/7 customer support.

HOW TO HOST YOUR WEBSITE

Hosting a website requires a few basic steps, including:

- Choose a hosting provider: There are various hosting providers available, such as Bluehost, HostGator, SiteGround, and many others. You can compare the pricing, features, and customer support of each provider and choose the one that best suits your needs and budget.
- Choose a hosting plan: Once you've selected a hosting provider, you'll need to choose a hosting plan. Most providers offer different plans based on the features and resources you need, such as storage space, bandwidth, email accounts, and more.
- Register a domain name: A domain name is your website's address, such as www.example.com. You can purchase a domain name from a domain registrar, such as Namecheap or GoDaddy.
- Connect your domain name to your hosting account: You'll need to update your domain's DNS settings to point to your hosting provider's servers. You can usually do this through your domain registrar's control panel.
- Upload your website files: You can use an FTP client, such as FileZilla, to upload your website files to your hosting account. Alternatively, many hosting providers offer website builders or one-click installation tools for popular CMS platforms like WordPress.
- Test your website: Once you've uploaded your website files, you'll want to test your website to make sure everything is working correctly. You can do this by visiting your website's URL in a web browser.
- Launch your website: Once you're satisfied with how your website looks and functions, you can officially launch it by making it live for the public.

It's important to note that the specific steps may vary slightly

depending on the hosting provider and website platform you choose.

CONCLUSION

This book takes you on a comprehensive journey through the fundamental pillars of modern web development. By diving deep into HTML, CSS, and JavaScript, this book equips you with the knowledge and skills necessary to create dynamic, interactive, and visually appealing websites.

Throughout the book, you've learned how HTML provides the structure and semantic markup of web pages, allowing you to organize and present content effectively. CSS has empowered you to transform these pages into captivating and visually stunning experiences, with the ability to customize layouts, colors, typography, and more.

With JavaScript, you've explored the world of interactivity and dynamic behavior, enabling you to add functionality, handle user interactions, and manipulate page elements in real-time. By mastering JavaScript, you've gained the power to create engaging user experiences and build web applications that go beyond static content.

Moreover, you've discovered the importance of responsive design, ensuring that your websites adapt flawlessly to various screen sizes and devices. This emphasis on responsiveness and user experience has become essential in today's mobile-driven world, and you're now equipped with the techniques and best practices to deliver seamless experiences across all platforms.

From the foundational concepts to more advanced topics, such as DOM manipulation, event handling, and AJAX, this book has provided you with a solid understanding of the core web development technologies. You've learned to harness the potential of HTML, CSS, and JavaScript to unleash your creativity and bring your visions to life on the web.

As you continue your journey in web development, remember that learning is an ongoing process. Stay curious, keep experimenting, and embrace new technologies and frameworks that emerge in this rapidly evolving field. By combining your knowledge of HTML, CSS, and JavaScript with a growth mindset, you'll be well-positioned to tackle any web development challenge that comes your way.

Now, armed with the knowledge gained from this book, go forth and create remarkable web experiences that inspire, engage, and make a lasting impact in the digital world. Happy coding!

INDEX

A

abstraction, 216

Abstraction, 216

addition, 229, 239, 289, 345

AJAX, 322, 323, 331, 333

alt, 7, 14, 20, 33, 72, 74, 78, 165, 176, 177, 188, 189, 201, 202

alternative, 14, 20, 33, 72

animation, 123, 124, 125, 126, 127, 128, 129, 133

animations, 3, 63, 64, 94, 123, 126, 127, 133, 138, 140, 159, 210

Animations, vi, 123, 124, 140

anonymous, 306, 338

API, 299, 300, 310, 319, 322, 323, 324, 325, 329, 331, 333

APIs, 8, 322, 323, 325, 331

app, 215, 305, 325, 340, 346, 350

applications, 210, 211, 212, 213, 214, 215, 216, 301, 322, 323, 331, 342, 345, 369, 370

apps, 210, 211, 212, 345

arguments, 264, 290, 293, 295, 302, 309, 312, 339

arithmetic, 229, 239

array, 224, 235, 236, 237, 273, 274, 275, 279, 310, 339

arrays, 223, 232, 233, 235, 238, 273, 338, 339, 342, 344

Arrays, 224, 235, 238, 273, 274

article, 4, 11, 75

assistive, 10, 73, 74

Async/await, 319, 320

asynchronous, 212, 308, 309, 310, 311, 313, 318, 319, 339

Asynchronous, 212, 308, 322, 331

Atom, 9, 217

attribute, 3, 5, 6, 20, 21, 25, 29, 31, 32, 33, 35, 36, 39, 41, 43, 44, 45, 46, 47, 48, 49, 51, 52, 53, 58, 59, 62, 70, 72, 74, 98, 135

attributes, 2, 13, 14, 20, 25, 33, 37, 39, 41, 42, 49, 51, 52, 53, 57, 79, 109, 134

Attributes, iii, 2, 13, 14, 20

audio, 2, 3, 7, 11, 38, 39, 40, 73

authentication, 323

Azure, 369

B

background, 308, 309, 327, 328, 334, 354, 355, 356, 364, 365

Backticks, 225

bandwidth, 371, 372

BitDegree, 210

block, 223, 254, 255, 256, 257, 258, 259, 260, 264, 276, 277, 278, 279, 320, 332, 338, 342, 364

block-level, 223

blocks, 24, 33, 159, 254, 264, 308, 319, 343

Bluehost, 371

blueprint, 289, 297, 338

body, 12, 13, 15, 16, 17, 18, 80, 82, 83, 85, 91, 118, 136, 141, 142, 146, 147, 152, 154, 164, 166, 175, 178, 179, 187, 190, 191, 200, 202, 203

Booleans, 223, 230

border, 55, 99, 108, 109, 110, 111, 112, 113, 114, 115, 143, 144, 150, 154, 156, 166, 179, 182, 183, 190, 192, 196, 202, 206

borders, 3, 55, 57, 60, 62, 107, 138

box model, 110, 111, 112, 113, 115

box-sizing, 111, 166, 179, 190, 202

braces, 233, 254, 255, 264, 282, 340, 341

bracket, 226

brackets, 235, 273

Brackets, 9

break, 260, 261, 262, 343

Brendan Eich, 210, 211

browser, 3, 5, 6, 8, 11, 12, 15, 16, 17, 18, 19, 21, 29, 34, 36, 39, 53, 63, 66, 70, 92, 99, 100, 117, 118, 124, 137, 138, 160, 210, 211, 212, 213, 214, 215, 217, 299, 302, 372

budget, 369, 371

button, 14, 42, 47, 48, 51, 52, 126, 127, 142, 144, 147, 150, 152, 153, 154, 156, 178, 182, 183, 185, 190, 196

C

C++, 213, 214, 216, 259

calculator, 350, 351, 354

callback, 308, 309, 310, 311, 312, 313, 314, 318

Callback, 309, 311

Canvas, iv, 7, 63, 64, 65, 66

case-sensitive, 213

catch(), 316, 317, 332

Chaining, 341

class, 213, 289, 290, 291, 292, 293, 295, 297, 298, 300, 326, 338, 351, 362, 363

classes, 214, 216, 289, 294, 297, 337, 341, 343

Classes, 289, 291, 337, 338

client, 63, 66, 67, 71, 72

Client-side, 212, 213, 215

closures, 264

Cloud, 369

Coalescing, 341

code, 3, 5, 6, 7, 8, 10, 11, 16, 19, 21, 39, 46, 47, 48, 49, 51, 52, 53, 55, 56, 58, 59, 64, 73, 77, 79, 80, 82, 83, 84, 91, 94, 96, 98, 99, 100, 101, 103, 107, 114, 117, 122, 123, 130, 131, 132, 133, 134,

136, 137, 138, 139, 141, 142, 145, 147, 151, 154, 163,166, 169, 187, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 225, 254, 255, 256, 257, 258, 259, 260, 262, 263, 264, 268, 269, 276, 277, 278, 279, 280, 282, 283, 293, 300, 301, 308, 309, 318, 319, 320, 321, 337, 339, 340, 341, 342, 343, 344, 345, 346

Code, 217, 220, 325, 326, 329, 333, 334, 335, 346, 347, 350, 353, 356, 362, 364, 366

Code Runner, 220

coding, 219, 343, 345

collections, 273

colon, 233

color, 3, 5, 55, 59, 62, 66, 73, 74, 94, 95, 97, 98, 99, 100, 104, 106, 107, 108, 109, 113, 125, 126, 127, 128, 129, 131, 132, 133, 142, 143, 144, 148, 149, 150, 155, 156, 160, 166, 167, 179, 180, 182, 183, 191, 192, 193, 196, 197, 203, 204, 205, 206, 207, 208

columns, 51, 54, 55, 56, 57, 58, 59, 60, 120, 121, 122

command, 218, 219, 220

compare, 244, 247, 371

comparison, 244

compatibility, 8, 124, 137, 171

compiled, 213, 214

compile-time, 214

complex, 211, 213, 215, 216, 223, 249, 309, 344, 350

concatenate, 227

condition, 249, 250, 251, 252, 253, 254, 255, 256, 257, 259, 274, 276, 277, 278

Conditional, 254

conditions, 249, 250, 251, 254, 258, 263, 325

configure, 220

connection, 71, 72

connections, 71, 172

consistency, 52, 101, 118

console, 225, 226, 228, 229, 231, 233, 234, 235, 236, 237, 238, 239, 241, 242, 243, 244, 245, 246, 247, 249, 250, 251, 253, 254, 255, 256, 258, 259, 262, 264, 265, 267, 268, 269, 270, 271, 274, 275, 276, 278, 279, 280, 283, 285, 286, 288, 290, 293, 296, 297, 298,302, 303, 304, 310, 311, 312, 314, 315, 317, 320, 323, 324, 325, 330, 332, 335, 340

constructor, 282, 289, 290, 292, 293, 294, 295, 296, 297, 298

Constructors, 294

Containerization, 370

content, 1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 23, 24, 25, 32, 38, 54, 63, 67, 72, 73, 74, 75, 76, 77, 78, 79, 80, 82, 85, 94, 95, 96, 104, 105, 110, 111, 112, 113, 116, 121, 124, 129, 131, 137, 141, 145, 148, 151, 152, 154, 157, 158, 162, 164, 168, 169, 171, 172, 175, 187, 192, 193, 200, 204, 205, 206

contrast, 213, 214, 215, 308, 342

controls, 36, 39, 40, 74, 119

converted, 324

create, 1, 2, 4, 7, 8, 11, 13, 14, 15, 20, 24, 25, 26, 28, 29, 31, 33, 41, 42, 45, 46, 47, 48, 50, 51, 54, 56, 61, 62, 63, 64, 65, 66, 67, 69, 71, 72, 79, 91, 93, 94, 95, 96, 104, 105, 110, 113, 115, 118, 120, 123, 126, 127, 128, 129, 130, 140, 157, 159, 160, 161, 162, 166, 171, 174, 187

C**ross-platform**, 8

cryptographic, 6

CSS, i, iv, v, vi, vii, 1, 2, 3, 8, 9, 12, 13, 25, 38, 55, 57, 59, 62, 66, 86, 87, 88, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 129,131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 145, 147, 151,

154, 157, 159, 160, 162, 163, 166, 169, 170, 171, 172, 174, 179, 187, 190, 199, 202, 210, 213, 300, 307, 326, 334, 346, 347, 350, 353, 362, 364

custom, 51, 132, 135, 136, 140, 163, 171

D

Data types, 222

debug, 138, 139

Debugging, vii, 138

declare, 223, 224, 225, 229, 231, 233, 235, 273, 319

dependencies, 219

Deployment, 369

Destructuring, 337, 338, 344

developers, 1, 2, 6, 8, 10, 11, 12, 21, 66, 67, 68, 94, 104, 105, 110, 115, 118, 120, 121, 136, 140, 159, 160, 162, 171, 210, 211, 212, 216, 232, 322, 323, 337, 341, 342, 343, 345, 370

development, 210, 211, 215, 216, 217, 219, 220, 301, 322, 323, 345, 369

devices, 212, 369

digital, 2

direction, 116, 124, 125, 129, 143, 182, 193, 205

directory, 30, 35, 136, 218, 219, 220

display, 5, 11, 54, 70, 75, 119, 120, 121, 129, 131, 143, 148, 154, 156, 167, 168, 169, 180, 181, 182, 184, 192, 193, 195, 204, 205, 206, 207

div, 3, 13, 14, 18, 70, 71, 74, 112, 114, 115, 119, 125, 128, 129, 142, 152, 153, 154, 165

Docker, 370

document, iii, 2, 4, 6, 12, 13, 14, 15, 16, 17, 18, 19, 21, 23, 33, 35, 38, 64, 65, 93, 94, 95, 99, 100, 101, 102, 103, 105, 116, 117, 118, 119, 121, 137

DOM, 299, 300, 302, 304, 305, 307, 322

DOM API., 299, 300

DOM tree, 299

domain, 371, 372

download, 217

DreamHost, 371

duration, 124, 125, 126, 127, 129

dynamic, 2, 6, 63, 64, 66, 93, 133, 159, 210, 211, 212, 213, 322, 323, 331

Dynamic, 213, 214

dynamically, 210, 213, 214, 215, 333

E

ECMAScript 2015,, 337

editions, 211

editor, 8, 9, 14, 16, 17, 33, 217, 218, 220

element, 273, 274, 279, 299, 300, 301, 302, 303, 304, 305, 306, 310, 333

elements, 2, 3, 5, 6, 7, 11, 13, 14, 15, 16, 17, 20, 33, 36, 39, 41, 42, 54, 55, 57, 58, 64, 65, 70, 72, 73, 74, 75, 91, 93, 94, 95, 96, 97, 98, 99, 100, 103, 104, 105, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 121, 123, 129, 131, 133, 134, 138, 158, 159, 163, 170, 173, 273, 274, 299, 300, 302, 303, 304, 305, 306, 307, 329, 333

embed, 225, 338

embedding, 3, 7

encapsulate, 289, 341, 343

encapsulation, 216

enclose, 24

Error, 312, 317, 320, 332, 343, 360

errors, 311, 316, 319, 339, 342, 343, 344

ES6, 214, 284, 337, 338, 339

Event, 212, 215, 301

event-driven, 212, 215

events, 212, 215, 303, 309

examples, 222, 224, 232, 272, 323, 342

execute, 254, 256, 257, 260, 276, 278, 279, 308, 310

executing, 211, 308, 309

execution, 212, 215, 279, 339

exercises, 1, 79, 174, 345

Exercises, 345

Express.js, 219

expression, 5, 52

expressions, 225, 239, 338

Expressions, 5

extensions, 210

Extensions, 220

external, 13, 95, 101, 102, 103, 134, 135, 166

F

Facebook, 212

features, 2, 3, 5, 7, 8, 9, 52, 54, 63, 79, 121, 139, 140, 151, 159, 171, 210, 212, 216, 337, 338, 341, 342, 345, 370, 371, 372

Fetch, 323, 324, 329, 331, 333

FileZilla, 372

finish, 308, 309

Flexbox, vi, 115, 116, 129, 130

floating-point, 223, 229

fonts, 93, 135, 136, 138, 159, 162

footer, 3, 4, 11, 57, 75, 77, 90, 147, 150, 151, 165, 169, 178, 183, 190, 197, 202, 208

form validation, 210

forms, 5, 11, 41, 46, 48, 51, 52, 79, 170

frameworks, 213, 215, 345

FTP, 372

full-stack, 210

function, 210, 213, 214, 223, 264, 265, 266, 267, 268, 269, 270, 271, 272, 279, 280, 283, 286, 295, 297, 298, 302, 303, 304, 306, 309, 310, 311, 312, 314, 319, 320, 338, 339, 340, 344, 348, 349, 356, 358, 360, 366, 367, 368

functional, 12, 79, 93, 95, 173, 211, 212, 215

functions, 233, 264, 279, 282, 283, 284, 286, 294, 308, 309, 311, 312, 313, 318, 319, 337, 338, 342, 343, 344, 372

Functions, 264, 265, 266, 268

G

games, 210

GCP, 370

gender, 233, 234

generic, 343

GitHub, 210

Global, 269, 342

GoDaddy, 372

Google, 211, 212, 370

Google's, 211

greater than, 245, 247, 248, 254, 255, 256, 258, 259

H

hardware, 369

head, 12, 13, 15, 16, 18, 79, 80, 82, 85, 100, 101, 103, 141, 145, 146, 152, 164, 174, 175, 187, 199, 200

header, 3, 4, 10, 11, 56, 75, 77, 85, 86, 146, 148, 164, 165, 166, 168, 169, 175, 179, 187, 188, 192, 200, 203

heading, 2, 10, 12, 19, 23, 33, 73

height, 37, 65, 94, 105, 106, 111, 112, 114, 115, 119, 127, 128, 134, 135, 160, 161, 168, 191, 193, 203, 205, 208

high-level, 210

hitcounter, 213

HostGator, 371

hosting, 210, 369, 370, 371, 372

Hostinger, 371

HTML, i, iii, iv, v, vii, 1, 2, 3, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 28, 29, 32, 33, 34, 35, 38, 45, 47, 48, 50, 51, 54, 57, 59, 61, 64, 65, 72, 74, 75, 79, 80, 82, 86, 87, 93, 94, 95, 96, 97, 98, 100, 101, 102, 103, 104, 105, 106, 107, 110, 112, 117, 119, 120, 121, 122, 123, 126, 130, 133, 134, 137, 138, 139, 141, 145, 151, 159, 164, 170, 171, 174, 187, 199, 210, 213, 300, 301, 302, 307, 325, 333, 346, 350, 362

HTML5, iii, iv, v, 2, 3, 5, 7, 8, 9, 12, 13, 35, 38, 41, 52, 63, 64, 65, 66, 67, 69, 70, 71, 73, 77

HTTP request, 309, 310, 311, 312, 313

I

ID, 25, 65, 98, 139, 299, 300, 301, 302

IDE, iii, 8, 9, 220

identifiers, 213

idiomaticity, 214

images, 2, 6, 7, 11, 13, 14, 17, 18, 23, 33, 35, 64, 70, 72, 74, 78, 84, 95, 119, 123, 133, 157, 158, 171, 172, 174

indentation, 343

inheritance, 213, 214, 216, 291

inherits, 291

init, 219

inline, 5, 95, 115, 119, 137, 156, 180

inner, 271, 272

Install, 217, 219, 220

instruction, 308

integers, 223, 282

intelligent, 254

interact, 215, 301, 322, 323

interaction, 210

interactions, 210, 213, 301

interactive, 2, 3, 6, 8, 14, 63, 64, 66, 67, 73, 74, 93, 94, 123, 133, 140, 171, 210, 211, 212, 213, 302, 322

interacts, 119

interface, 299

internet, 212

interpreted, 210, 213, 214

Interpreter, 213

J

Java, 211, 213, 214, 216, 259

JavaScript, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 222, 223, 224, 225, 229, 231, 232, 233, 235, 239, 244, 249, 254, 259, 260, 264, 269, 273, 279, 282, 283, 289, 291, 294, 297, 299, 302, 308, 309, 313, 319, 322, 323, 329, 331, 335, 337, 338, 341,

342,343, 344, 345, 346, 347, 350, 356, 362, 366

json, 219, 319, 320, 323, 324, 329, 332, 335

JSON, 311, 312, 319, 321, 323, 324, 331, 332

K

keyframes, 123, 124, 127, 128, 129, 138

keyword, 222, 223, 224, 264, 268, 283, 286, 289, 290, 291, 293, 294, 295, 298, 319, 321, 338, 339, 340, 342

keywords, 213, 222, 224, 239, 338

Kubernetes, 370

L

language, 2, 20, 64, 65, 93, 210, 211, 212, 213, 214, 215, 299, 337, 341, 342

languages, 213, 214, 215, 216, 259

layout, 1, 59, 93, 94, 95, 96, 104, 110, 115, 116, 117, 120, 121, 129, 139, 157, 160, 161, 162, 170, 173, 174

libraries, 213, 217, 345

licensed, 220

links, iv, 6, 11, 12, 13, 17, 18, 23, 25, 29, 31, 32, 64, 74, 75, 76, 78, 84, 95, 96

Links, 6, 29, 33

list, 13, 14, 25, 26, 27, 28, 33, 38, 42, 46, 47, 48, 61, 62, 70, 148, 149, 155, 167, 179, 181, 191, 203

lists, 11, 13, 14, 15, 18, 23, 25, 26, 28, 29, 33, 54, 61, 62, 72, 74, 79, 106

literals, 225, 227, 337, 338, 344

LiveScript, 211

Local, 270

logged, 302, 320, 324, 325

logic, 213

logical, 231, 249, 252, 253, 341

loop, 274, 275, 276, 277, 278, 300

LTS version, 218

M

Main, 4

manipulate, 225, 232, 274, 300, 302

manipulation, 299, 307

margin, 99, 110, 111, 112, 113, 114, 115, 116, 131, 138, 142, 143, 144, 147, 148, 149, 150, 154, 155, 156, 166, 167, 168, 169, 179, 180, 181, 182, 183, 185, 190, 192, 193, 194, 195, 196, 202, 204, 205, 206, 207

markup, 2, 9, 10, 11, 20, 64, 65, 73, 74, 75, 137

match, 1, 52

meta, 12, 13, 73, 77, 79, 80, 82, 85, 141, 145, 152, 164, 174, 175, 187, 199, 200

Meter, 5, 6

methods, 211, 212, 215, 216, 223, 225, 273, 274, 282, 283, 284, 286, 287, 289, 290, 291, 293, 294, 295, 297, 298, 299, 300, 303, 308, 315, 337, 338, 339, 341, 344

Microsystems, 211

minlength, 53

mobile, 210, 211, 212, 215

modern, 1, 7, 52, 66, 67, 68, 71, 94, 116, 124, 132, 138, 157, 210, 211, 331, 333

modules, 337, 338, 339, 343

multimedia, 2, 3, 7, 79

multiple, 57, 58, 66, 94, 96, 101, 104, 137, 139, 157, 158, 223, 263, 266, 273, 309, 315, 320, 338, 339, 341

multiplication, 229, 239

N

name, 211, 214, 222, 224, 225, 228, 233, 234, 238, 264, 265, 266, 282, 283, 289, 290, 293, 295, 296, 297, 298, 302, 305, 313, 314, 315, 324, 325, 326, 330, 333, 341, 346, 351, 362, 372

Namecheap, 372

names, 213, 290, 343

nav, 11, 75, 76, 85, 131, 146, 148, 149, 164, 165, 167, 169, 175, 179, 180, 183, 187, 188, 192, 200, 203, 204

navigation, 4, 11, 73, 74, 75, 76, 130, 131, 151, 159, 162, 169, 170, 173

Nested, iii, 28

Netflix, 212

Netscape, 210, 211

Node.js, 210, 211, 215, 217, 218, 219

NOT, 252, 253

notation, 226, 233, 235, 282, 283, 287, 288, 289, 338

npm, 217, 219

Numbers, 223

O

object, 210, 212, 214, 215, 216, 223, 233, 234, 262, 274, 282, 283, 284, 286, 289, 290, 293, 294, 295, 296, 298, 299, 300, 313, 314, 315, 319, 321, 324, 331, 332, 338, 339, 341, 342

objects, 211, 212, 214, 215, 216, 223, 225, 232, 233, 235, 238, 282, 283, 286, 289, 294, 295, 296, 297, 322, 337, 338, 339, 342, 343, 344

Objects, 214, 223, 233, 282, 283

OOP, 215, 216

operating, 217, 218

operator, 227, 239, 240, 241, 242, 243, 245, 246, 250, 252, 253, 339, 341

operators, 229, 231, 239, 244, 247, 248, 249, 338, 339, 341

option, 47, 48, 49, 50, 51

ordered, 13, 14, 25, 26, 27, 33, 61, 62

Ordered, iii, 26, 61

outer, 271, 272

P

package, 217, 219

packaging, 370

padding, 99, 110, 111, 112, 113, 114, 115, 138, 142, 143, 144, 148, 149, 150, 154, 155, 156, 166, 167, 179, 180, 181, 182, 183, 184, 191, 192, 194, 195, 196, 197, 203, 205, 206, 208

pages, 1, 2, 3, 10, 11, 12, 13, 14, 20, 29, 30, 54, 63, 72, 73, 78, 84, 93, 94, 96, 100, 101, 104, 105, 110, 118, 120, 137, 138, 160, 210, 211, 213, 215, 322, 323

paragraph, 2, 10, 19, 20, 22, 24, 25, 33, 97, 116

parameter, 266, 339

parameters, 264, 265, 266, 290, 293, 338, 339

pattern, 5, 52

performance, 211, 212, 309, 370

PHP, 215

pixels, 37, 65, 94, 95, 98, 106, 107, 113, 114, 115, 117, 119, 135, 160, 161, 169

Placeholders, 4

platforms, 210, 369, 370, 372

polymorphism, 216

portfolio, 345

practical, 222, 232, 345

Preferences, 220

Preload, 4

presentation, 9, 93, 94, 95, 96

problems, 211, 342

professional, 86, 93, 141

programmers, 211, 282

programming, 210, 211, 212, 213, 214, 215, 216, 222, 249, 254, 259, 264, 282, 294, 299, 308, 309, 313, 337, 338

programs, 211, 299

Progress, 3, 5

project, 218, 219, 220, 345

Project, 345

Promises, 313, 318, 319, 321, 338, 339

properties, vi, 94, 97, 98, 99, 104, 105, 106, 107, 108, 109, 110, 111, 113, 114, 115, 116, 118, 121, 122, 123, 124, 125, 126, 127, 129, 132, 136, 137, 138, 140, 161, 211, 212, 214, 215, 216, 223, 225, 233, 234, 282, 283, 284, 289, 290, 291, 293, 294, 295, 297, 298,300, 303, 338, 339, 341, 342

property, 4, 5, 6, 96, 97, 99, 108, 109, 110, 111, 113, 114, 115, 116, 117, 118, 119, 120, 121, 124, 126, 127, 128, 129, 131, 133, 134, 161, 233, 234, 275, 283, 291, 293, 298, 301, 303, 304, 306, 315, 324, 332, 341

protocol, 71

prototypal, 213, 214

prototype, 214, 295, 296, 297, 298

Prototype, 213

prototypes, 294, 297

published, 211

purposes, 210, 323

Q

queries, 157, 158, 160, 161, 162, 163, 171, 174, 183

R

Registration, v, 82

Relative, iv, 30, 117

Removes, 274

represents, 230, 233, 273, 299, 313

requests, 308, 325, 331, 333

required, 33, 44, 52, 53, 80, 82, 83, 91, 134

reset, 51, 138

Response, 319, 332

responsive, vii, viii, 1, 66, 67, 73, 93, 94, 96, 115, 116, 120, 121, 129, 133, 140, 151, 157, 158, 159, 161, 163, 170, 171, 172, 173, 174, 186, 187, 199

Responsive, i, vii, viii, 1, 140, 157, 158, 159, 170, 171, 172, 186, 199, 200

Return Values, 268

reusable, 211, 212, 215, 216, 264, 343

Reverse, 6

rows, 51, 54, 55, 57, 60, 120, 121, 122

Ruby, 213, 215

Run, 220, 221

runtime, 213, 214, 217

Ryan, 211

S

Scalable, vii, 64, 133, 134

scenarios, 212, 322, 345

scope, 223, 264, 269, 270, 271, 272, 338, 345

Scope, 269, 270

screen, 1, 10, 11, 14, 46, 70, 71, 73, 74, 80, 82, 93, 94, 96, 116, 118, 121, 129, 140, 157, 158, 160, 161, 162, 163, 164, 168, 169, 171, 172

scripting, 210, 211, 212, 214, 215, 299

section, 4, 11, 15, 23, 32, 75, 77, 86, 87, 88, 90, 100, 101, 103, 146, 147, 149, 151, 165, 167, 168, 169, 175, 176, 177, 178, 180, 184, 188, 189, 190, 200, 201, 202, 232, 254, 294

Section, 4, 32, 192, 194, 195

secure, 343

security, 6, 211

semantic, 9, 10, 11, 72, 74, 75

server, 8, 11, 41, 47, 48, 63, 71, 72, 210, 215, 219, 309, 322, 323, 331, 332, 370

servers, 210, 212, 331, 345, 369, 372

setAttribute, 301

Shortly, 211

sidebar, 220

sizes, 1, 18, 86, 93, 94, 96, 104, 116, 118, 129, 140, 157, 158, 160, 161, 162, 163, 169, 171, 172

skill, 7, 79, 87, 137

skills, 1, 79, 84, 86, 87

smartphones, 1, 8, 160, 174

Sockets, v, 71

software, 216, 322, 323, 369

SSL, 371

statement, 254, 256, 257, 259, 260, 262, 276, 314, 315, 340

statements, 213, 223, 230, 254, 263, 344

storage, 370, 371, 372

string, 225, 226, 262, 338

Strings, 223, 225, 227

style, 2, 6, 25, 38, 41, 54, 55, 57, 59, 62, 65, 93, 95, 97, 98, 100, 102, 104, 105, 106, 113, 136, 137, 141, 146, 148, 149, 152, 155, 159, 160, 167, 175, 179, 181, 187, 191, 200, 203

Styling, iv, 59, 60

subclass, 216, 291

subheading, 24

Sublime, 9, 16, 17, 217

submit, 11, 41, 42, 51, 80, 83, 142, 178, 190, 196

subtraction, 229, 239

supports, 211, 212, 215

SVG, iv, vii, 63, 64, 65, 66, 79, 133, 134, 135

switch, 213, 254, 259, 260, 262

synchronous, 308

syntax, 214, 220, 222, 224, 232, 254, 256, 257, 260, 276, 277, 279, 280, 284, 293, 319, 337, 338, 340, 342

T

Tables, iv, 54, 57, 59

tablets, 1, 8, 160, 174

tag, 2, 3, 4, 6, 7, 12, 13, 18, 20, 22, 23, 24, 33, 36, 38, 39, 45, 46, 47, 48, 49, 50, 51, 98, 100, 101

tags, 2, 3, 4, 9, 11, 12, 13, 14, 16, 17, 18, 20, 21, 23, 25, 28, 32, 33, 48, 50, 52, 61, 73, 77, 79

tech, 8

techniques, 1, 54, 96, 116, 140, 157, 171, 172, 174, 187, 199

technologies, 1, 8, 10, 64, 73, 74, 157, 159

technology, 64

templates, 210

terminal, 219, 220

Time, 5, 6, 177

title, 12, 13, 15, 16, 18, 23, 24, 80, 82, 85, 141, 146, 152, 164, 175, 187, 200

trends, 139

Twitter, 212

type, 2, 5, 6, 12, 14, 18, 36, 39, 42, 44, 45, 46, 47, 51, 52, 53, 62, 66, 80, 82, 83, 88, 115, 117, 118, 119, 141, 142, 152, 160, 161, 164, 177, 178, 190, 196

types, 213, 214, 216, 222, 223, 225, 254, 269, 273, 282

U

undefined, 224, 285, 341, 342

unordered, 13, 14, 25, 26, 28, 33, 61

URL, 3, 14, 20, 21, 29, 33, 41, 103, 135, 136, 323, 332, 372

URLs, iv, 30, 73, 78

user, 1, 3, 4, 5, 7, 8, 11, 14, 21, 22, 29, 32, 36, 38, 39, 41, 43, 44, 45, 46, 47, 48, 51, 52, 63, 64, 66, 67, 68, 69, 70, 71, 94, 118, 119, 123, 124, 133, 157, 158, 160, 161, 162, 172, 174

users, 1, 5, 6, 9, 10, 11, 29, 32, 41, 45, 46, 48, 51, 54, 70, 71, 72, 73, 74, 77, 124, 140, 158, 162

V

validation, 5, 52, 53

value, 4, 20, 21, 42, 44, 47, 48, 49, 66, 67, 80, 83, 96, 97, 99, 114, 116, 118, 119, 132, 133

variable, 213, 214, 222, 224, 225, 229, 231, 262, 269, 270, 271, 272, 273, 332, 341

variables, 213, 214, 222, 224, 239, 269, 270, 271, 337, 338, 342, 343, 344

Versatility, 215

video, 2, 3, 4, 35, 36, 37, 73, 210

videos, 4, 7, 11, 35, 72, 74

Videos, iv, 4, 35

Visual Studio, 9, 16, 83, 217, 220

W

web, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 19, 20, 23, 25, 29, 32, 33, 34, 36, 37, 39, 40, 41, 45, 46, 48, 51, 52, 54, 63, 66, 67, 68, 69, 70, 71, 73, 74, 75, 76, 77, 79, 93, 94, 95, 96, 99, 100, 101, 104, 105, 110, 112, 113, 115, 117, 118, 120, 123, 124, 125, 133, 135, 137, 138, 140, 141, 144, 151, 156, 157, 158, 160, 165, 170, 171, 173, 185, 197, 208, 210, 211, 212, 213, 215, 217, 219, 299, 301, 322, 323, 331, 333, 345, 371, 372

Web, i, iii, iv, v, vii, 2, 8, 11, 63, 66, 67, 68, 69, 71, 94, 133, 140, 171

web page, 212, 215, 299, 301, 322

web pages, 11, 12, 93, 104, 138

webpage, 307, 322, 325, 333

website, viii, 4, 5, 6, 11, 30, 32, 41, 72, 73, 74, 78, 94, 95, 96, 100, 101, 104, 115, 124, 133, 135, 136, 157, 158, 159, 161, 162, 171, 172, 173, 174, 187, 199, 217, 371, 372

websites, 1, 3, 5, 6, 7, 8, 72, 79, 93, 94, 96, 100, 101, 104, 133, 140, 141, 157, 158, 159, 162, 171, 172, 210, 213

WebSocket, 71, 72

WebStorm, 9

width, 6, 37, 60, 65, 80, 82, 85, 111, 112, 113, 114, 115, 116, 119, 127, 128, 134, 135, 141, 143, 145, 149, 152, 154, 155, 160, 161,

164, 168, 169, 175, 183, 185, 187, 191, 196, 200, 208

X

XML, 322, 323, 331